Prime Numbers Considered Useful: Ontology Encoding for Efficient Subsumption Testing

Davy Preuveneers
Yolande Berbers

Report CW 464, October 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Prime Numbers Considered Useful: Ontology Encoding for Efficient Subsumption Testing

Davy Preuveneers
Yolande Berbers

Report CW 464, October 2006

Department of Computer Science, K.U.Leuven

Abstract

Multiple inheritance hierarchies are frequently used for the classification of concepts into a taxonomy, to model software by organizing classes into an inheritance hierarchy, for querying object-oriented databases, for knowledge representation, policy enforcement, and subtyping of service interfaces for safe composition and substitution. All these areas apply hierarchies and share the same concern of being able to compute inheritance or subsumption relationships efficiently. In this report, we elaborate on encoding multiple inheritance hierarchies for representing subsumption relationships between concepts defined in ontologies to enable efficient matching mechanisms for context-aware service discovery, composition and substitution, and other application domains.

We present several encoding techniques originating from the programming languages domain for the subtyping of classes in multiple inheritance hierarchies and argue why these encoding techniques are not feasible for achieving efficient subsumption testing in ontologies. We developed a prime-based encoding technique for subsumption of concepts and properties that outperforms several well-known ontology reasoners for this particular problem, both in terms of having a compact representation of the encoding and being able to compute the subsumption relationship efficiently. Our algorithm also offers an interesting alternative for the class subtyping encoding techniques, as it yields a new way of compaction without the need for changing old conflicting codes during incremental encoding.

Keywords : subsumption, ontology, prime number, hierarchical encoding

Prime Numbers Considered Useful: Ontology Encoding for Efficient Subsumption Testing

Davy Preuveneers Yolande Berbers
Katholieke Universiteit Leuven, Belgium,
{davy.preuveneers, yolande.berbers}@cs.kuleuven.be

October 19, 2006

Abstract

Multiple inheritance hierarchies are frequently used for the classification of concepts into a taxonomy, to model software by organizing classes into an inheritance hierarchy, for querying object-oriented databases, for knowledge representation, policy enforcement, and subtyping of service interfaces for safe composition and substitution. All these areas apply hierarchies and share the same concern of being able to compute inheritance or subsumption relationships efficiently. In this report, we elaborate on encoding multiple inheritance hierarchies for representing subsumption relationships between concepts defined in ontologies to enable efficient matching mechanisms for context-aware service discovery, composition and substitution, and other application domains.

We present several encoding techniques originating from the programming languages domain for the subtyping of classes in multiple inheritance hierarchies and argue why these encoding techniques are not feasible for achieving efficient subsumption testing in ontologies. We developed a prime-based encoding technique for subsumption of concepts and properties that outperforms several well-known ontology reasoners for this particular problem, both in terms of having a compact representation of the encoding and being able to compute the subsumption relationship efficiently. Our algorithm also offers an interesting alternative for the class subtyping encoding techniques, as it yields a new way of compaction without the need for changing old conflicting codes during incremental encoding.

Contents

1 Introduction **5**

2 Use cases of subsumption testing **6**
- 2.1 Subtyping in programming languages . . . 6
- 2.2 Subsumption in taxonomies . . . 8
- 2.3 Subsumption in ontologies . . . 9
- 2.4 Subsumption in service discovery, composition and substitution . 9
- 2.5 Subsumption in policy enforcement . . . 11
- 2.6 Efficient subsumption under an open world view . . . 12

3 Definitions, terminology and notations **14**

4 Related work on encoding algorithms for inheritance hierarchies **16**
- 4.1 Bit vector-based hierarchical encodings . . . 16
  - 4.1.1 Ait-Kaci, Boyer, Lincoln and Nasr . . . 17
  - 4.1.2 Caseau . . . 18
  - 4.1.3 Krall, Vitek and Horspool . . . 19
  - 4.1.4 Caseau, Habib, Nourine and Raynaud . . . 20
  - 4.1.5 van Bommel and Beck . . . 20
- 4.2 Interval-based hierarchical encodings . . . 21
  - 4.2.1 Agrawal, Borgida and Jagadish . . . 21
  - 4.2.2 Constantinescu and Faltings . . . 22
  - 4.2.3 Zibin and Gil . . . 23

5 Using primes for hierarchy encoding **23**
- 5.1 Compact representation with prime numbers . . . 24
- 5.2 Conflict-free and fast incremental encoding with prime numbers . 29
- 5.3 Incremental encoding and reuse in ontologies . . . 29
- 5.4 Incremental encoding and equivalent classes . . . 31

6 Optimizing encoding and subsumption testing **32**
- 6.1 Optimizing the encoding length of the representation . . . 33
  - 6.1.1 Heuristic 1: Semi-random top-down order . . . 35
  - 6.1.2 Heuristic 2: Class with most descendants . . . 35
  - 6.1.3 Heuristic 3: Leaf class with most ancestors . . . 36
  - 6.1.4 Heuristic 4: Leaf class with the largest minimum code . . 37
- 6.2 Optimizing the subsumption test . . . 39
  - 6.2.1 The bit vector length of both classes . . . 40
  - 6.2.2 The personal prime number of both classes . . . 40
  - 6.2.3 Principal prime number analysis . . . 40

7 Experimental evaluation and validation **43**
- 7.1 Subtyping algorithms . . . 43
- 7.2 Subsumption with ontology reasoners . . . 44
- 7.3 Modeling other ontological features with prime numbers . . . 45

8 Conclusions and future work **46**

1 Introduction

In recent years, many researchers have addressed the issue of subsumption in several different application domains. Subsumption involves the possibility to incorporate or classify something under a more general or more comprehensive category. Taxonomies are one of the simplest classifications of entities as they impose a strict hierarchy with a single category at the top, i.e. the root class, and more specific subcategories defined by single inheritance of mutually exclusive and unambiguous subclasses. More complex organizations of entities, such as polyhierarchies or latices, have classes with multiple parents. Other topologies may allow multiple classes without parents. In general, the hierarchy that is imposed by the subsumption relationship can be viewed as a directed acyclic graph (DAG).

Subsumption testing is a common way of matching an entity's property with the one that was initially requested. When modeling these properties in a directed acyclic graph, one is sure that an appropriate entity has been found if a path exists in the graph from the requested property to the one that was provided. In that case it is said that the type of the requested property subsumes the provided one. This subsumption test is very common and known by different names in various application domains. As the size of the directed acyclic graph depends on the application domain at hand and varies from several nodes to several thousands of nodes, traversing the graph may become an expensive operation. First of all, the subsumption test cannot be done in constant time and backtracking is required for multiple inheritance hierarchies. Therefore, more efficient subsumption tests are required.

In this paper, we focus on encoding techniques for fast subsumption testing within ontologies. Ontologies come from the knowledge representation field and are used to structure an established vocabulary of terms and concepts within a specific domain of interest, and this in a way similar to multiple inheritance hierarchies. In short, ontologies provide a specification of a conceptualization [19]. The use of ontologies has been a very active field of research in the context-aware computing domain and in the Semantic Web community where they are applied for the semantic modeling of context [37, 8, 20, 42, 31] and services [25, 34, 3, 11, 17] to automate their discovery, composition and invocation in pervasive computing environments. Context, by definition [12], is any information that can be used to characterize the situation of entities (i.e. whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Therefore, context involves relevant information that needs to be described in a semantic model to facilitate the sharing and the uniform understanding of the collected information. The services, that were previously only described by their public invocation interfaces, are now enriched with machine-interpretable semantics by referring to and reasoning on established ontologies that encapsulate the meaning of the service. This is illustrated in the WSDL 2.0 standard, that now not only supports the use of XML Schema, but also provides standard extensibility features for using, e.g., classes in OWL ontologies to define web service input and output data types.

The encoding presented in this paper is proposed to counter one of the major disadvantages of ontologies: the reasoning tools and technologies underlying the semantic modeling with ontologies are particularly expensive in terms of mem-

ory and processing time. They are not designed with the limited and distributed computational resources within a highly dynamic and pervasive computing environment in mind. Some of the ideas for fast subsumption testing in ontologies presented in this paper are based on the principles of the encoding techniques that are used in compiler design and subtyping in programming languages. The same techniques cannot be directly applied to ontologies due to a different view on the classes and the concepts at hand, i.e. a closed world assumption on the one hand versus an open world assumption on the other hand. This difference will be explained later. The support for efficient incremental and conflict-free encoding for very large multiple inheritance hierarchies under an open world assumption was the main reason why we have developed a new approach for encoding subsumption relationships in ontologies. Conflict-free encoding ensures the freely reusing of existing ontology encodings and extending encoded concepts without leading to false subsumption results. The other key requirements for the encoding algorithm are to achieve a compact bit representation of the classes and properties and to provide a fast subsumption testing algorithm.

In section 2 we provide background information of several application domains where subsumption testing is used and discuss why existing encoding techniques cannot be directly applied to ontologies. Section 3 describes the terminology that is used throughout the rest of the paper. Section 4 provides an overview of related work on encoding techniques for multiple inheritance hierarchies and efficient subsumption testing. Section 5 outlines the basic principles of our prime-based hierarchical encoding. Several optimizations for achieving a compact representation and an efficient subsumption test are discussed in section 6. In section 7 we conduct several experiments with real world ontologies and compare the efficiency with existing ontology reasoners and other encoding techniques. We end with conclusions and future work in section 8.

2 Use cases of subsumption testing

Multiple inheritance hierarchies play an important role in classifications of entities [16], modeling classes in object-oriented languages [6], safe substitution of components with one another [24], and in many other domains. In this section we mainly focus on subtype testing or subsumption in the domains of programming languages and knowledge representation, and outline their different characteristics that spur the need for optimized subsumption techniques for each of these areas.

2.1 Subtyping in programming languages

One research area where subsumption is deeply explored, is the domain of object-oriented programming languages. For a multiple inheritance hierarchy of classes or types, subsumption boils down to knowing if a class or type is an ancestor of another one [7, 40]. See Figure 1 for a multiple inheritance hierarchy of persons within a university. All classes inherit from the most general class *Person*. One often refers to this subsumption relationship as *subtyping*, *type inclusion* or *type conformance*. Note that for some languages with interfaces, such as Java, a difference is made between subtyping and subclassing. Java only supports single inheritance of classes, but allows multiple inheritance of

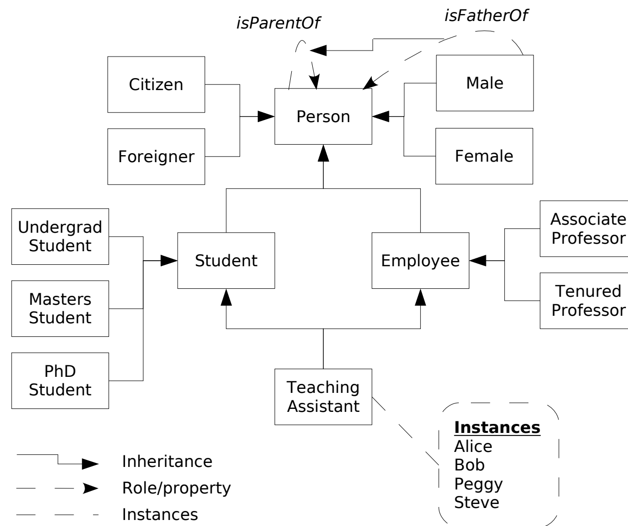

Figure 1: A multiple inheritance hierarchy of persons within a university

interfaces, where interfaces provide the means for subtyping but not for code reuse. For the sake of simplicity we make no distinction between subtyping and subclassing. A more correct statement would be that subclassing implies subtyping.

For *statically* typed languages, such as C++ and Java, the multiple inheritance hierarchy is used for type checking at compile time. Based on the declared type of the variables, the compiler computes the apparent type of a variable in any expression during the semantic analysis of the source. The compiler then guarantees that for any initialization of a variable or any evaluation of an expression that the actual type is always a subtype of the apparent type. This is illustrated with the following Java code:

```
Person p = new TeachingAssistant();
p = "Hello World";                          // Incompatible types!
```

The compiler will detect that *String* is not a subtype of *Person* and will end with a compilation error. For programming languages that support *dynamic* typing of classes, such as Smalltalk and Python, the second assignment is perfectly valid as variables can acquire different types depending on the execution path. Type checking is done at runtime, and is illustrated in the following Python code:

```
p = TeachingAssistant()
p = "Hello World"                           # p is of type string
p = p + 5                                   # Incompatible types!
```

In this code fragment the type of variable p changes from *Teaching Assistant* to string. However, when trying to concatenate a string with an integer, a type error will be detected at runtime. Whether a string can be concatenated with an integer depends on the language having a *strong* or a *weak* typing. Python

has a strong typing and thus rejects the third assignment, whereas PHP allows such statements due to its weak typing and implicit type conversion. Dynamic typing is often achieved by tagging all values with a type and checking the type before using the value in an expression.

Whether the type checking takes place at compile time or at runtime, the type inclusion test should be handled very quickly. To this aim efficient constant time type inclusion tests have been developed [6, 23, 41, 39, 7, 44] to ensure performance without sacrificing the expressiveness of the programming language. In a following section we will discuss in detail several encoding techniques for type hierarchies using bit vector and interval representations.

2.2 Subsumption in taxonomies

In [43], Woods defines taxonomies as structured conceptual descriptions for representing terms or concepts at different levels of generality, where the subsumption relationship between each pair of terms is based upon inheritance. Note that a taxonomy typically has a single inheritance structure. This is illustrated in the following vehicle taxonomy:

- ↪ vehicle
  - ↪ motor vehicle
    - ↪ automobile
      - ↪ car
      - ↪ taxi
    - ↪ railway vehicle
      - ↪ metro
      - ↪ train
      - ↪ tram
    - ↪ craft
      - ↪ aircraft
      - ↪ hovercraft
      - ↪ spacecraft
    - ↪ motorcycle
    - ↪ bus
    - ↪ truck
    - ↪ snowmobile
  - ↪ pedal powered vehicle
    - ↪ bicycle
    - ↪ velomobile

A request for *railway vehicle* also retrieves the terms *metro, train* and *tram.* General terms subsume specific ones. More specifically, a term subsumes itself, terms that are more specific, and all equivalent terms (or synonyms). When searching for a specific term, any term that is subsumed by the requested term will match.

Taxonomies are commonly used for classification purposes. The ACM Computing Classification System [1] is a well-known taxonomy for the categorization of publications according to various keywords. The taxonomy structures many computing related research fields into an inheritance hierarchy with 4 levels of

[1] http://www.computer.org/portal/pages/ieeecs/publications/author/ACMtaxonomy.html

specialization. This report could be classified under *H.3.3 Information Search and Retrieval*, but also under a more specialized category *H.3.4.d Performance evaluation*. These non-overlapping categories are both subsumed by their most general category *H Information Technology and Systems*.

2.3 Subsumption in ontologies

Subsumption also plays an important role as a reasoning task in knowledge representation [13] with ontologies. An ontology is a formal explicit description of *concepts* (often called *classes*) in a domain of discourse, properties describing various features and attributes of the concept (also called *slots* or *roles*), and restrictions on slots (sometimes referred to as *facets* or *role restrictions*) [28]. Concepts can be defined as specializations of other concepts. Therefore, ontologies impose a structure similar to multiple inheritance hierarchies describing vocabularies of useful terms for non-ambiguous exchange of information. Ontologies are considered more expressive than taxonomies as concepts can be distinguished by axioms and definitions stated in a formal language, such as description logics [19]. Common logical constructors include the *intersection*, *union* and *negation* of concepts, as well as the *existential* and *universal value restrictions* on roles.

A knowledge base is the combination of the ontology together with the individual *instances* of the concepts in the ontology. As such, a concept refers to the set of individuals that can be categorized under this concept. Roles denote a binary relationship between concepts. It is said that a concept A subsumes another concept B if the set of individuals represented by concept A is a superset of the set represented by concept B. The subsumption relationship can be specified for roles as well. For example, in Figure 1 the *isParentOf* role of the concept *Person* subsumes the *isFatherOf* role of the concept *Male*. A related reasoning task is *instance checking*. It tries to verify the membership of a given individual as an instance of a given concept. The individual *Alice* is an instance of the concept *Employee*, but not of the concept *Associate Professor*.

The W3C has endorsed the Web Ontology Language (OWL) [26] as a recommendation for representing concepts and their interrelationships. OWL provides three sublanguages – OWL Lite, OWL DL, and OWL Full – with varying degrees of expressiveness, with or without computational guarantees for completeness and decidability.

2.4 Subsumption in service discovery, composition and substitution

In our research on context-aware discovery of devices and composition of services, ontologies are used to model contextual information, the capabilities of a device, and to provide syntactic and semantic specifications of pervasive computing services. OWL-S [5] is a standard service ontology implemented in the OWL language that helps to fulfill this need. Other semantic modeling languages for web services have been proposed [34, 11, 3], but we will focus on the OWL-S language for illustration purposes.

OWL-S consists of a set of basic classes and properties for declaring and describing services for the Semantic Web, enabling users to discover, select and

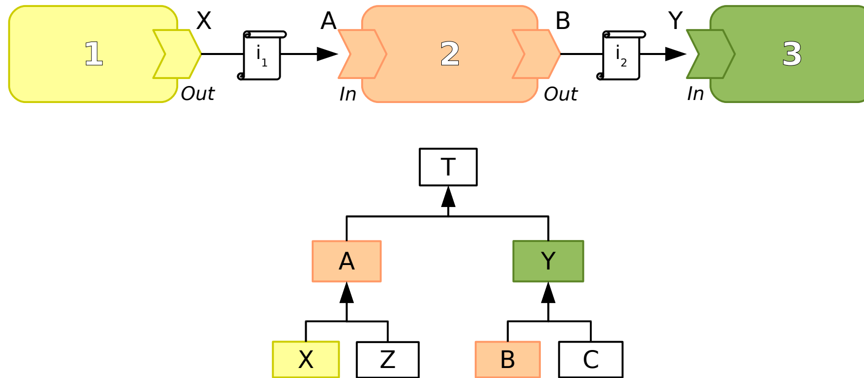

Figure 2: Matching input/output interfaces in a service composition

compose services in a more automated way. Some service properties of the OWL-S ontology that serve well the purpose of subsumption include the *hasInput*, *hasOutput*, *serviceClassification* and the *serviceProduct* properties:

- **hasInput, hasOutput**: These properties can be used for checking a possible service composition at the syntax level. For example, during service discovery we may be looking for a service that will need to interact with another given service. Therefore, the interfaces of the cooperating services should match. This is illustrated in the service composition in Figure 2. Service 2 delivers its output of type B to service 3 and requires input of type A from service 1. Any instance i_1 (here of type X) whose type is subsumed by type A will match. Likewise should the type of instance i_2 (here of type B) be subsumed by type Y. We use subsumption in a similar way to find devices with specific capabilities.

- **serviceClassification, serviceProduct**: These two properties refer to service and product catalogues defined outside the OWL-S specification. One of these classifications is the Universal Standard Products and Services Classification (UNSPSC) [16] of which a small subset is shown in Figure 3. Another one is the North American Industry Classification System (NAICS)[2]. The UNSPSC classification has more than 20000 entries and each of the entries is identified by a specific code and classified according to a *segment, family, class* and *commodity* title. Each of these 4 levels can be used to restrain the service discovery protocol. However, a service category by itself does not provide enough information to ensure correct interoperability.

During service discovery and composition, subsumption is used for matching service interfaces, checking the classification of a service in a standardized service catalogue, and verifying other contextual constraints. The subsumption relationship can also be used to make sure that services can be replaced with one another. This principle is also known as the *Liskov substitution principle* [24] in object-oriented design and declares that objects with a reference to

[2]http://www.census.gov/epcd/www/naics.html

↪ Cleaning Equipment and Supplies
↪ Commercial and Military and Private Vehicles and their Accessories and Components
↪ Communications and Computer Equipment and Peripherals and Components and Supplies
  ↪ Communications and computer supplies
    ↪ Audio recording media
    ↪ Data storage media
    ↪ Multimedia storage
      → Compact disc CD cases
      → Diskette storage
      → Multimedia drawers
      → Multimedia towers
      → Multimedia trays or organizers
      → Zip mailers
  ↪ Hardware and accessories
  ↪ Software
↪ Defense and Law Enforcement and Security and Safety Equipment and Supplies
↪ Distribution and Conditioning Systems and Equipment and Components

Figure 3: A small subset of the UNSPSC taxonomy

a base class must be able to use objects of a derived class. Liskov stated this as follows:

> *What is wanted is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.*

Given the service composition in Figure 2, this means that service 2 can be replaced with another service 2' if all the requested interfaces and properties of service 2' are subsumed by those of service 2.

2.5 Subsumption in policy enforcement

Ontologies are also used to specify policies [22, 29] because OWL ontologies are more expressive and the inferencing strengths of a description logic reasoner simplify computationally complex operations, such as checking whether one policy is covered by another, whether two policies contradict, etc. In [29], the authors show how typical policy issues map to inferencing problems:

- **Policy inclusion**: A policy A includes a policy B if policy A subsumes policy B, i.e. instances of policy B are also instances of policy A.
- **Policy equivalence**: Two policies A and B are equivalent if policy A subsumes policy B and policy B subsumes policy A.
- **Policy incompatibility**: An individual can never be an instance of two different policies A and B at the same time. This occurs when two requirements in policies A and B contradict.

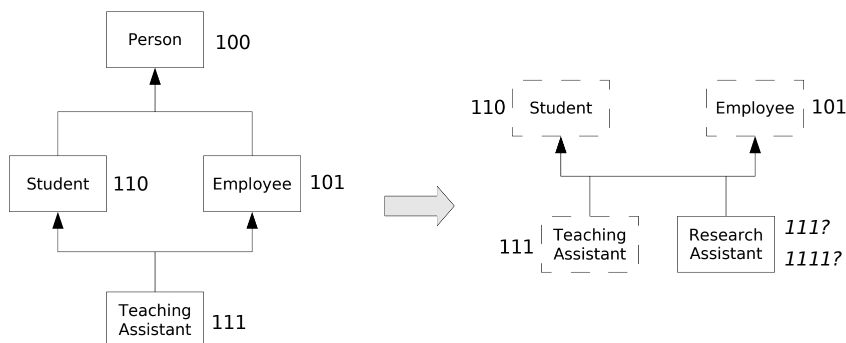

Figure 4: Bit vector encoding and open/closed world assumption

- **Policy incoherence**: A policy can never have an instance. This may happen when two requirements in the same policy contradict.
- **Policy conformance**: A given individual is an instance of a given policy, if it complies with all the requirements of the policy.

It is clear that the concepts *policy* and *class* are very much related. The requirements specified in a policy are somewhat similar to the properties of a class. An individual meets a policy if it fulfills certain requirements. Meeting a policy is very similar to the instance checking problem mentioned earlier. For this application domain, subsumption is a commonly used operation as well.

2.6 Efficient subsumption under an open world view

In the previous sections we discussed how subsumption can be used for a wide variety of reasoning and matching tasks. As ontologies provide a more expressive language to describe relationships between concepts and roles, we use OWL ontologies exclusively for the modeling of classes and properties in multiple inheritance hierarchies. Though many OWL reasoners, such as Racer [21], OWL Pellet [30] and FaCT++ [38], support subsumption queries, their counterparts for subtype testing in object-oriented programming languages are often more efficient in terms of performance and memory consumption:

- **Compact bit representation:** The data structures for representing subsumption relationships are small. Each class is usually represented as a bit vector, with various optimizations for reusing bit positions for different classes. However, other representations, such as intervals, are used as well. Examples of such encodings will be given in a following section.
- **Fast matching:** Bit vector representations allow for efficient subsumption or subtype testing by means of binary AND and OR operations. Testing for subsumption can therefore be done in constant time and result in predictable matching speeds.

There are several reasons why some subtype encoding techniques for testing type inclusion are not useful for representing subsumption within ontologies.

One of the reasons is the difference in the view on the classes and concepts involved, i.e. subsumption under a *closed world* assumption within a compiler versus an *open world* assumption in ontologies. Another reason is the scalability to very large ontologies with support for conflict-free incremental encoding to easily reuse previously defined and encoded concepts.

- **Closed world versus open world assumption:** If a compiler knows all the classes in a program (closed world assumption), it can do a more efficient and compact encoding of the classes by reusing bit positions. The compiler can assume that no other classes are being subsumed if such classes do not exist. OWL assumes an open world view, which means that no assumption is made on the completeness of the provided information. This is illustrated by means of a simple encoding in Figure 4. Consider the multiple inheritance hierarchy on the left with 4 classes and a 3 bit encoding. A class with bit vector X subsumes a class with bit vector Y only if the following statement holds: X AND $Y = X$ (or X OR $Y = Y$).

  Assume under the open world view that in a new ontology the codes for *Student* and *Employee* classes are reused and that a new class *Research Assistant* is introduced which also inherits from both *Student* and *Employee*, as shown on the right. There exists no code Z for this new class for which the binary AND test succeeds for the parent classes *Student* and *Employee*, but not for the *Teaching Assistant* class. The subsumption test would result in a false positive. The *Teaching Assistant* class needs to be re-encoded by adding a new bit.

- **Scalability:** To overcome the previous concern, one simple solution is to introduce a new bit position for each new class. However, this does not scale well for ontologies with several thousands of classes [10]. Both the compact bit representation and the fast matching are affected by this solution. Therefore, other representations should be considered that allow compaction and efficient matching under an open world assumption.

- **Incremental encoding:** Some encoding techniques have support for incremental encoding, but require the re-encoding of conflicting codes whenever a subsumption test results in a false positive. Again, for large ontologies this re-encoding is not feasible. Reuse of ontologies and the concepts defined therein is one of the main goals of ontologies, and thus re-encoding would occur rather often whenever conflicts arise due to new inheritance relationships. As such, an incremental encoding technique without the need for re-encoding but with support for compaction is highly desirable. Preferably, the encoding should only be carried out once in advance and later on be reused by as many parties as possible.

In this paper we present an encoding mechanism for multiple inheritance hierarchies that overcomes the previous issues. Though it does not cover all the expressiveness and capabilities of a DL reasoner, it is perfectly usable for the subsumption scenarios that were outlined earlier in section 2. The encoding itself is based on the multiplication of co-prime numbers and being able to quickly detect whether one number divides another one. We will first describe the basics of this encoding technique before going into detail on several optimizations for both the subsumption test and the encoding. Our technique outperforms the

subsumption test of the previously mentioned ontology reasoners, both in terms of having a compact representation of the encoding and being able to detect subsumption efficiently.

3 Definitions, terminology and notations

Due to the very nature of subsumption being applied in various application domains, each using its own terminology, it is of utmost important that similar concepts and terms are explained and properly defined. This is the purpose of this section. All application domains have in common that the basic structure that is used to model subsumption can be represented with a multiple inheritance hierarchy, as illustrated in Figure 1.

Hierarchy: This is the partial ordered set of classes that reflects the subsumption relationships. The structure may conform to a strict hierarchy with only single inheritance, or to more complex structures, such as a lattice or a polyhierarchy, supporting multiple inheritance of classes. The hierarchy is represented with a symbol χ.

Class: Refers to a node in the hierarchy χ, which can be a *class* or a *type* in an object-oriented language, or a *concept* in an ontology. A class reflects a set of instances. Names of classes always start with a capital letter: *Person, Student, Female, ...*

Instance: The instantiation of a class can be a runtime *instance* of a type or class, i.e. an object, or an *individual* of a concept. Each instantiation of the same class has the same set of properties, though their values may differ. The name of an instance always starts with a small letter and possibly has an index when used in a collection of instances with similar names: $Student = \{ s_1, s_2, ..., s_n\}$

Property: This is an attribute of a class that defines a relationship from one instance to another. It is also called a *role* in ontology terminology. In this definition, primitive datatypes are not considered as properties because the subsumption relationship is not defined for them. The name of a property starts with a small letter and is preceded with a dot and the name of either the class or the instance: *Person.isParentOf*, *male.isFatherOf*

Child: A class $A \in \chi$ is a child of a class $B \in \chi$ if and only if class A has a *direct* connection with class B and inherits from class B in the hierarchy. A *Leaf* class is a special child in the hierarchy that has no children of its own. The child relationship is denoted as: $A <:_d B$

Parent: A class $A \in \chi$ is a parent of a class $B \in \chi$ if and only if class B is a child of class A. In a single inheritance hierarchy, each class has at most one parent. In a polyhierarchy, classes can have multiple parents. A *Root* class is a parent in the hierarchy with no parents of its own. The parent relationship is denoted using the reversed child notation as: $B <:_d A$.

Descendant: A class $A \in \chi$ is a descendant of class $B \in \chi$ if and only if it is either a child of class B or a descendant of one of the children of class B.

This recursive definition of descendant is denoted as: $A <: B$. A child is a *direct* descendant, hence the subscript d in the child notation.

Ancestor: A class $A \in \chi$ is an ancestor of class $B \in \chi$ if and only if class B is a descendant of class A. Note that in a single inheritance hierarchy there is only one path between a class A and a descendant/ancestor class B. The ancestor relationship is denoted as: $B <: A$

Equivalence: Two classes A and B in the hierarchy χ are defined equivalent if both classes have the same set of instances or individuals at all times. The equivalence relationship is denoted as: $A \equiv B$

Subsumption: Refers to the reflexive, transitive and anti-symmetric relationship of classes in the hierarchy χ, which states that a class $A \in \chi$ is subsumed by a class $B \in \chi$ if and only if the set of instances of class A are also included in those of class B. This is the same as stating that class B subsumes class A. The subsumption relationship, which is also referred to as *type inclusion*, is denoted as: $\gamma(A) \subseteq \gamma(B)$. Note that: $\gamma(A) \subseteq \gamma(B) \Leftrightarrow A \equiv B \vee A <: B$.

Supersumption: This is the inverse of subsumption. A class $A \in \chi$ is supersumed by class $B \in \chi$ if and only if class B is subsumed by class A. This is the same as stating that class B supersumes class A, or that class A subsumes class B. The supersumption relationship is denoted using the reversed subsumption notation as: $\gamma(A) \supseteq \gamma(B)$.

Gene: A gene is a unique symbolic or numeric identifier that is used to define a subsumption relationship between two classes. If a class A inherits from a class B, then it inherits all the genes of class B. Some classes introduce a new personal gene that it only shares with its descendants. As such, a class may have a personal gene and inherited genes. An inherited gene is a personal gene of an ancestor. Note that the gene metaphor not always applies as in some encoding algorithms genes are passed along from the leaves to the roots. The personal gene of class A is represented as g_A.

Code: A class is encoded into a compact representation that allows efficient subsumption testing. Bit vectors and intervals are commonly used representations. For bit vector representations, the encoding is often determined by first assigning to each gene a distinguishing bit position in the bit vector, and then for each class setting the corresponding bits of its genes to 1. The code of a class A is represented as $\gamma(A)$.

Given these definitions and notations for specifying relationships between classes in a multiple inheritance hierarchy, we declare the following functions for a class $A \in \chi$:

$$\begin{array}{rcl}
parents(A) & = & \{\ B \in \chi \mid A <:_d B\ \} \\
children(A) & = & \{\ B \in \chi \mid B <:_d A\ \} \\
ancestors(A) & = & \{\ B \in \chi \mid A <: B\ \} \\
descendants(A) & = & \{\ B \in \chi \mid B <: A\ \} \\
roots(A) & = & \{\ B \in ancestors(A) \mid parents(B) = \emptyset\ \} \\
leaves(A) & = & \{\ B \in descendants(A) \mid children(B) = \emptyset\ \}
\end{array}$$

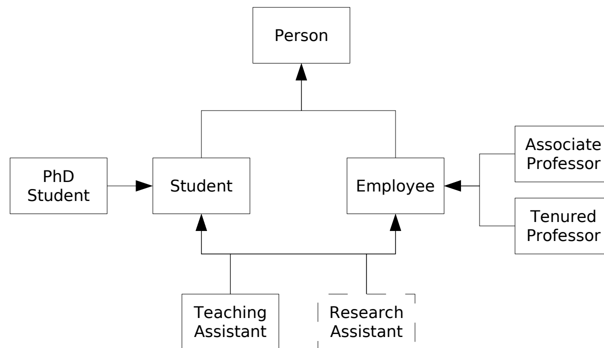

Figure 5: A multiple inheritance hierarchy encoding example

We now continue to describe several encoding techniques for efficient subsumption testing.

4 Related work on encoding algorithms for inheritance hierarchies

In this section we will mainly focus on hierarchical encoding techniques originating from the programming languages domain, as many algorithms have been proposed in the past for efficient type inclusion testing. The most common representation for such encodings use bit vectors, but other representations, such as intervals, are used as well. Where appropriate, we will highlight the benefits or drawbacks of a particular encoding, especially in the presence of multiple inheritance and whether efficient incremental encoding is supported. We will illustrate the encodings with the example in Figure 5 where possible. For incremental encoding, we assume that a new child (e.g., *Research Assistant*) is added, while possibly being a parent itself for existing classes.

The most simple but a rather naive approach to determine if a class X subsumes a class Y is to use the inheritance hierarchy directly by starting at class Y and going upwards to find class X as an ancestor of class Y. The disadvantages of this method are that subsumption cannot be done in constant time and that the approach only works for single inheritance hierarchies. Backtracking is required for polyhierarchies and is very expensive for very large multiple inheritance hierarchies [10].

4.1 Bit vector-based hierarchical encodings

A straightforward constant time subsumption technique that supports multiple inheritance encodes the inheritance hierarchy using a binary matrix of size $n \times n$, with n the number of classes in the hierarchy. The array is filled with 0's, with 1's on the diagonal running from position $(1, 1)$ to (n, n) and on positions (i, j) if class i is an ancestor of class j. This representation requires n^2 bits, can be used for multiple inheritance, and is easily expandable for new classes (such as the *Research Assistant* class) by adding a new row and column and filling in the

Class	Binary Matrix	Ait-Kaci	Caseau	Krall	Agrawal	van Bommel
Person	11111111	11111	000000	000000	[1-7]	0
Student	10100011	10011	000001	000001	[1-3]	1
Employee	10011101	11101	000010	000010	[4-6],[2-2]	10
Associate Professor	00001000	01000	000110	000110	[4-4]	110
Tenured Professor	00000100	00100	001010	001010	[5-5]	1010
PhD Student	00000010	00010	000101	000101	[1-1]	10001
Teaching Assistant	00000001	00001	010011	010011	[2-2]	1000011
Research Assistant	10000000	10000	100011	100011	impossible	100011

Table 1: Encodings of the multiple inheritance hierarchy

1's as before. The results are shown in Table 1. The underlined bits illustrate the modifications after the incremental encoding of the *Research Assistant* class. Subsumption can be checked in constant time with binary AND operations on the bit vectors:

$$\gamma(Person) \subseteq \gamma(Employee) \Leftrightarrow \gamma(Person) \text{ AND } \gamma(Employee) = \gamma(Employee)$$
$$\gamma(Employee) \not\subseteq \gamma(Student) \Leftrightarrow \gamma(Employee) \text{ AND } \gamma(Student) \neq \gamma(Student)$$

These equations can be rewritten with binary OR operations as follows:

$$\gamma(Person) \subseteq \gamma(Employee) \Leftrightarrow \gamma(Person) \text{ OR } \gamma(Employee) = \gamma(Person)$$
$$\gamma(Employee) \not\subseteq \gamma(Student) \Leftrightarrow \gamma(Employee) \text{ OR } \gamma(Student) \neq \gamma(Employee)$$

4.1.1 Ait-Kaci, Boyer, Lincoln and Nasr

A more compact encoding compared to the binary matrix representation was achieved by Ait-Kaci *et al.* [2]. It is a bottom-up approach: the encoding starts with selecting different bit positions for the leaf classes and continues with each parent by using the binary OR operation on the codes of its children. In gene terminology, each class passes along its personal and other genes along to its parents. New bit positions are only introduced when a parent has only one child in order to distinguish the parent from its child, and whenever a false positive subsumption result would arise. For a single inheritance tree with $k < n$ leaves and each parent having at least 2 children, only $k \times n$ bits are needed. Incremental encoding can be handled in a similar way as in the previous encoding by adding a new row and column. However, this only works if the new class is a leaf class. If, for example, a new class X would be inserted as a sibling of the class *Person* with the same inheriting subclasses, then another extra bit would have to be introduced to re-encode the class *Person*, as otherwise the sibling classes X and *Person* would inherit from one another. Note that in this case any ancestor of the class *Person* would have to be re-encoded as well. Moreover, for a new non-leaf class the ancestors of its children should be checked for conflicting codes. Assume we have the following hierarchy and codes:

$$D = 011 \leftarrow \{A = 001, B = 010\} \quad E = 110 \leftarrow \{B = 010, C = 100\}$$
$$F = 111 \leftarrow \{D = 011, E = 110\}$$

This representation means that class D has classes A and B as children, and that the encoding of class D is computed using the OR operation on the codes of its

Class	Code
Person	000000
Student	000001
Employee	000010
Associate Professor	010110
Tenured Professor	001010
PhD Student	000101
Teaching Assistant	100011
Research Assistant	000111

Table 2: Re-encoding conflicting classes

children. Then a new incrementally added root class $G \leftarrow \{A = 001, C = 100\}$ cannot have code 101 as this code would falsely assume that class G is subsumed by class F. If, as another example, class G would be incrementally added as $G \leftarrow \{A, B, C\}$, then distinguishing bit positions would be required for classes D, E and G, and all their ancestors would have to be re-encoded. The bit positions of the leaf classes correspond to those in the previous column to easily compare the results with the previous encoding.

4.1.2 Caseau

Caseau [6] proposed a top-down encoding algorithm that first transforms the hierarchy into a lattice. A lattice is a partially ordered set that defines for each pair of classes a *least upper bound* (LUB) and a *greatest lower bound* (GLB). The initial hierarchy in Figure 5 is already in lattice shape. The algorithm locates classes with only one parent (here: *PhD Student*, *Student*, *Employee*, *Associate Professor* and *Tenured Professor*) and then assigns a gene to each of them. A gene is a distinguishing bit and corresponds to a certain bit position that is set to 1 in the bit vector. Genes can be reused for classes that are unrelated due to other inherited genes, as illustrated in the third column of Table 1. The third bit position (from the right) is shared by the classes *PhD Student* and *Associate Professor*. Classes are encoded by using the OR operation on the codes of their parents. For larger structures, Caseau's encoding results in a more compact encoding than the one by Ait-Kaci [2]. When adding the *Research Assistant* class during incremental encoding, the hierarchy is no longer a lattice. A new virtual class X, of which both the *Teaching Assistant* and *Research Assistant* classes inherit, is inserted at the position of the *Teaching Assistant* to enforce a lattice structure. Now both the assistant classes have only one parent, i.e. the class X, and have become candidates for being assigned a gene. For this incremental encoding example, two genes, i.e. bit positions, had to be added into the encoding.

The incremental encoding can become even more complex. Assume that the *Research Assistant* class would inherit from the *PhD Student* class instead of the *Student* class. To compute the code for the *Research Assistant* class the OR operation is applied on the codes of its parents *PhD Student* and *Employee*. This results in the code: 0111. Now both the *Associate Professor* and *Teaching Assistant* classes would falsely subsume the *Research Assistant* class. This could be quickly resolved by again adding two extra bit positions to re-encode the two conflicting classes, and this would result in the encoding in Table 2. However, it now appears as if the *Associate Professor* class has been assigned

two bit positions. Caseau describes a more complex algorithm that reassigns safer bit positions to conflicting classes, with a more compact encoding for larger hierarchies in the end. The disadvantage is that the algorithm requires traversing for each conflicting class all the descendants and the children of all the ancestors of a class.

Moreover, ensuring a lattice structure may prove to be not so convenient for the encoding length either, especially for large hierarchies. Assume a classification of pizzas, of both the vegetarian and non-vegetarian kind, is encoded as a single class *Pizza* with the different types of pizzas as children of the *Pizza* class [3]. When a new class *Vegetarian Dish* is incrementally added and includes among its children n types of pizzas with vegetarian topping, then about $n^2/2$ additional classes would have to be introduced for lattice completion. Moreover, Krall *et al.* [23] have shown that Caseau's incremental encoding algorithm fails for certain hierarchies.

4.1.3 Krall, Vitek and Horspool

After encountering some issues with Caseau's encoding, Krall *et al.* [23] proposed another top-down algorithm that does not need to maintain the lattice structure. The algorithm itself is based on graph coloring. The authors propose to determine those classes that cannot use the same gene and construct a conflict graph with edges between each of two such classes. Before this conflict graph is constructed, the hierarchy may be balanced for classes with a large branching factor, i.e. with many children that each would require a different gene. This is done by splitting up the children in smaller groups and adding additional classes as parents of each group. This way more genes can be reused. The algorithm continues to mark all the classes in the hierarchy that require a gene. Define S as the set of classes with a single parent, M as the set of classes with multiple parents, and G as the set of classes which require a gene. G is initialized with the classes that only have a single parent: $G = S$. A class $n \in M$ is added to G if another class $m \in M$ exists, not subsumed by n, for which:

$$(1) \qquad ancestors(m) \bigcap G \subseteq ancestors(n)$$

The algorithm then continues by constructing the conflict graph for the classes in G. There is an edge between two distinct classes p and q if:

$$(2) \qquad q \in descendants(parents(p)) \quad \vee$$
$$(3) \qquad q \in ancestors(descendants(parents(p)) \setminus descendants(p))$$

After coloring the graph using heuristics, as graph coloring is NP-hard, the code for each class is computed as the union of all the genes for its ancestors and for itself. Each gene is assigned a specific bit position in the bit vector. For the example in Figure 5, the classes in G are: *Student, PhD Student, Employee, Tenured Professor* and *Associate Professor*. As the initial structure only includes one class with multiple inheritance, rule (1) is never used. The conflict graph looks like Figure 6. The encoding would be completely the same as the one of Caseau if all genes are assigned to the same bit positions. If the class *Research Assistant* is incrementally added, then both the classes *Teaching Assistant* and *Research Assistant* are added to the set G according to rule (1)

[3]See http://www.co-ode.org/ontologies/pizza/ for a more advanced ontology for pizzas

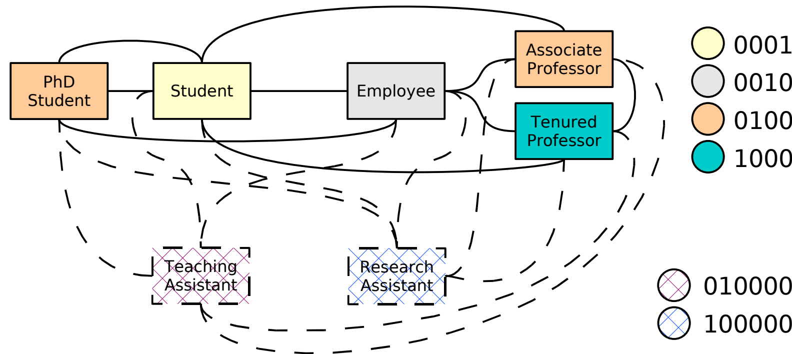

Figure 6: Conflict graph construction and coloring

and after coloring, two extra genes are required. The authors show for other examples that due to a good graph coloring algorithm and balancing the hierarchy, the code length can be reduced up to a quarter of the size of Caseau's encoding. The algorithm itself is not really suited for incremental encoding as it requires the reconstruction and recoloring of the conflict graph for new classes. Simple and fast heuristics could be used for updating the current coloring with a less optimal coloring as a result. Note that when a class' code is changed, that all its descendants will need to be re-encoded. The fastest solution would be to add new genes for new classes if their inherited code would conflict with existing classes, as already outlined for Caseau's method. The authors discuss further compaction algorithms in [41].

4.1.4 Caseau, Habib, Nourine and Raynaud

In turn, Caseau *et al.* [7] identified a shortcoming in the work of Krall [23]. Caseau provides an example of a hierarchy where a conflict edge is introduced between two classes according to rules (2) and (3) in Krall's algorithm. The authors improve the definition and the calculation of the conflict graph that avoids the false conflicts. This conflict graph is computed for the set of *join-irreducible elements* $J(P)$ of the partial order $P = (\chi, \leq)$. $J(P)$ is defined as the set of elements $j \in \chi$ for which the following statement holds:

$$\exists\, x \in \chi :\ x \notin descendants(j) \ \wedge\ ancestors(j) \subseteq ancestors(x) \bigcup \{x\}$$

The remainder of the algorithm is similar to the encoding technique of Krall: it includes conflict detection, coloring the conflict graph and preprocessing the hierarchy to an equivalent one with a smaller branching factor. The authors also show how the new algorithm can also be used for incremental encoding of new leaf classes. For each new leaf class, the coloring of the old conflict graph is preserved and the new join-irreducible elements are colored according to the new conflict graph.

4.1.5 van Bommel and Beck

In [39], van Bommel *et al.* examine several top-down incremental encoding methodologies, i.e. without recomputation of existing codes. For bit vector

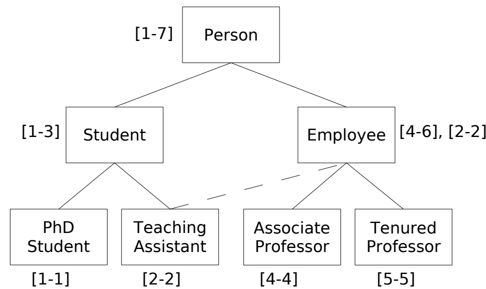

Figure 7: Interval encoding by Agrawal using a spanning tree

representations, the variable length encoding algorithm assigns a code to each new class by computing the binary OR operation on the codes of its parents. If the new class has only one parent, then a new bit is added to the front of the new class' code. In case of a conflicting code with those of its parents' descendants, a new bit is added to the front of the codes of the conflicting classes. The new bit is possibly propagated to their respective children.

Grouping of classes in the hierarchy is used in order to share bit positions in codes of classes that belong to different groups. Each group is assigned a distinct code and classes within a group are top-down encoded. Group combination is required when a newly added class inherits from classes from different groups. This is done by adding for each group a distinct bit to all the classes of the same group and merging the classes in a new group. The new group is also assigned a new code by computing the binary OR of the old groups' codes. The encoding shown in Table 1 is without grouping. The authors also propose a variant of the incremental interval-based hierarchy encoding of Agrawal [1].

4.2 Interval-based hierarchical encodings

Bit vectors are not the only representation used for efficient subsumption testing, intervals are a candidate as well. Two values constituting the interval are stored at each class in the hierarchy, with children of a class having non-overlapping intervals that fall inside the interval of its parent. A simple encoding technique traverses the tree in-order, labeling the starting point of the interval when entering the node and labeling the end point of the interval when leaving the node. This encoding only works for single inheritance hierarchies and incremental encoding is not possible either. A similar algorithm called *relative numbering* [33] traverses the tree in post-order and assigns an interval to a class using the lowest order number of its descendants and its own order number as the start and end point of the interval respectively.

4.2.1 Agrawal, Borgida and Jagadish

Agrawal *et al.* [1] proposed a variant of the latter algorithm in their *range compression* algorithm that supports multiple inheritance by first constructing a spanning tree of the hierarchy, encoding the spanning tree in post-order as explained previously, and adding additional intervals to some parent classes involved in multiple inheritance, i.e. with edges not in the spanning tree. This

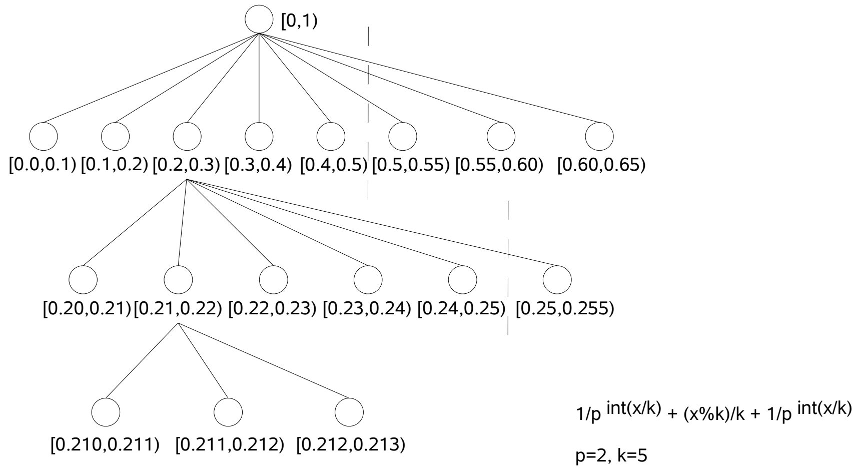

Figure 8: Interval encoding by Constantinescu

is illustrated in Figure 7. Subsumption of a class is then based on containment of its intervals in one of the intervals of an ancestor. This algorithm does not support incremental encoding but the authors propose to modify the encoding scheme by leaving gaps in the numbers used for classes in the spanning tree, and using these gaps for encoding new classes.

4.2.2 Constantinescu and Faltings

Constantinescu *et al.* [9] discuss an interval-based encoding algorithm for multiple inheritance hierarchies. The boundaries of the intervals are defined by floating point numbers. Intervals can be contained in one another, but are never overlapping. A class in a multiple inheritance hierarchy can have multiple intervals. The size of the intervals are defined by two parameters p and k. The root class of the hierarchy has an interval $[0, 1)$. The intervals in the levels below are divided by an exponential factor $1/p$. Each interval $1/p^i$ is then linearly divided by a factor k. To this aim the authors define a linear inverse function $linKinvexpP(x) = 1/p^{int(x/k)} + (x\%k)/k * 1/p^{int(x/k)}$ and use $p = 2, k = 5$ in their example. An example of such an encoding for a single inheritance graph is illustrated in Figure 8. Due to the limited representation of a 64-bit floating point number it is important to know how many classes can be represented with this encoding scheme. For parameters $p = 2, k = 5$ the authors are able to represent 1071 classes in a level, and 462 levels in the hierarchy. This should be sufficient for any incremental encoding of the hierarchy. The interval encoding by itself does not support the modeling of multiple inheritance. Therefore, the algorithm provides additional encoding support for multiple inheritance by converting a multiple inheritance hierarchy to single inheritance hierarchies, encoding and augmenting them with a mapping table that links each originally multiple inheriting class to one or more single inheriting classes.

4.2.3 Zibin and Gil

Zibin *et al.* [44] proposed another encoding algorithm that is based on relative numbering and PQ-trees. Without going into details on all the optimizations and compression schemes, we summarize that their PQ-encoding algorithm improves the encoding length, creation time, test time and instruction count of the *Near Optimal Hierarchical Encoding* algorithm of Krall *et al.* [23]. The main disadvantage though is that the encoding algorithm cannot be easily modified for incremental encoding, as the PQ-data structures do not support efficient updates.

5 Using primes for hierarchy encoding

In the previous section several encoding algorithms for efficient subsumption or subtype testing were discussed. As previously outlined, the major concern is support for efficient incremental and conflict-free encoding. While most algorithms provide a way to encode new classes without re-encoding the whole hierarchy, they often require going through the hierarchy to find and modify any conflicting codes. The only algorithm that does not suffer from this issue is the binary matrix method, but this method is the most expensive one with respect to the encoding length as no compaction of the representation is achieved. In this paper we propose an encoding that meets the following requirements:

- **Compact representation:** A compact representation means that all classes can be represented in the encoding without requiring a distinguishing bit position for each class. For variable length encoding, the representation can either be optimized in terms of either the total encoding length for all codes, or minimizing the length of the largest code of all classes.

- **Efficient subsumption:** Subsumption testing should be based solely on the encoding of the two classes at hand and not on any knowledge on the structure of the multiple inheritance hierarchy or on the codes of the other classes in the hierarchy. These constraints enable a hierarchy independent and constant time optimized implementation of the subsumption testing algorithm.

- **Conflict-free incremental encoding:** An efficient conflict-free incremental encoding method avoids conflicts rather than solving them and reduces the traversing of the hierarchy to a minimum in order to calculate the new code. Especially adding new leaf classes to the hierarchy should be handled only by looking at the codes of the parents.

Incremental encoding without conflicts requires that each class in the hierarchy has its own personal gene. If each gene is represented by means of a distinguishing bit position in the bit vector then compaction becomes a necessity for very large hierarchies, as otherwise n^2 bits are required for representing an inheritance hierarchy with n classes in a binary matrix. One way to reduce the total size of the representation is to use variable length bit vectors. However, the binary matrix also represents all ancestors in the bit vector of a specific class. If explicit knowledge of the ancestors is not required, then other compaction

mechanisms are possible. The problem statement can be defined as follows:

Assume for a set of classes in a hierarchy $\chi = \{C_1, C_2, ..., C_n\}$ and for a set of genes $G = \{g_1, g_2, ..., g_n\}$ that a function $\varphi : \chi \to G$ uniquely maps classes C_i to different genes g_i:

$$\forall\ C_i \in \chi : g_i = \varphi(C_i) \tag{1}$$

Define Γ as the set of genes that a class $A \in \chi$ inherits from its ancestors and equivalent classes $B \in \chi$, and its own personal gene g_A:

$$\Gamma(A) = \{\ g_A\ \} \ \cup\ \{\ g \in G \mid B \in \chi\ \wedge\ (A <: B\ \vee\ A \equiv B)\ \wedge\ g = \varphi(B)\ \} \tag{2}$$

If classes $X, Y \in \chi$ are equivalent, i.e. $X \equiv Y$, then the above definition ensures that $\Gamma(X) = \Gamma(Y)$. For an optimized encoding it would make sense to reuse the same genes for equivalent classes. Also, a function $\psi(\Gamma)$ is required that encodes a set of genes $\Gamma(C)$ to a compact representation $\gamma(C)$:

$$\forall\ C_i \in \chi : \gamma(C_i) = \psi(\Gamma(C_i)) \tag{3}$$

such that for $C_i \in \chi$ with encoding length $|\gamma(C_i)|$ (in bits):

$$max(\{\ |\gamma(C_i)|\ \}) \qquad \text{or} \qquad \sum\ |\gamma(C_i)| \qquad \text{is minimal}$$

for either optimized encoding length or optimized total encoding length respectively, and for which the subsumption test $subsumes(\gamma(X), \gamma(Y))$ for any pair of classes X and Y can be computed efficiently. Corollary, if classes $X, Y \in \chi$ are equivalent, then $\gamma(X) = \gamma(Y)$.

5.1 Compact representation with prime numbers

One observation of the binary matrix representation is that the matrix is sparse, i.e. it contains many 0's, especially when bit positions cannot be reused. The reason for this is that the number of classes is usually at least an order of magnitude larger than the average number of ancestors for a class in a hierarchy. This fact can be exploited by only encoding a reference to the ancestors in the code of a class. This can be done by assigning a number as personal gene $g_i \in G$ to each class $C_i \in \chi$, that is co-prime with all the other genes. The encoding function $\psi(\Gamma)$ can then be defined as the multiplication of a class's personal gene with the inherited genes of its ancestors and those of equivalent classes.

The genes $g_i \in G$ are pairwise co-prime if they have no common factor other than 1, or if the following statement holds:

$$\forall g_i, g_j \in G : gcd(g_i, g_j) = 1 \tag{4}$$

The factorization of each gene $g \in G$ as a product of prime powers $g = \pi_1^{e_1}...\pi_n^{e_n}$ for positive e_i and distinct primes $\pi_i \in P$ is unique. The index i of π_i does not denote an order in the set of primes P, only that two primes π_i and π_j with $i \neq j$ are different. When referring to the ith prime number, we use the following notation: $p_i \in P$, i.e. $p_1 = 2, p_2 = 3, p_3 = 5$ and so on. It is important to remark that as genes are pairwise co-prime, a prime π_i can only occur in the factorization of one gene. The encoding $\gamma(C_i)$ as defined in (3) is redefined as the product of the genes in $\Gamma(C_i)$:

$$\gamma(C_i) = \prod_j\ g_j \qquad with \qquad g_j \in \Gamma(C_i) \tag{5}$$

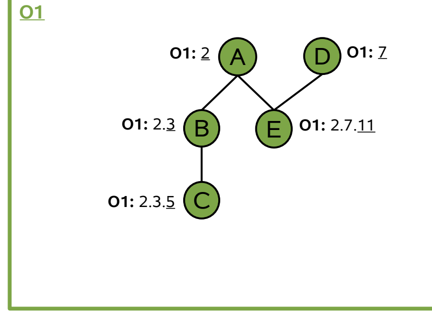

Figure 9: Hierarchy encoding using prime numbers

Note that genes $g \in G$ should not be prime numbers. For example, the genes in $G = \{76, 93, 145, 161, 169, 187\}$ are pairwise co-prime but are not pair numbers. However, since each prime $\pi \in P$ can only be present in the factorization of one gene, using primes as genes results in shorter encodings. The encoding function can thus be redefined as:

$$\gamma(C_i) = \prod_j g_j = \prod_j \pi_j \qquad \text{with} \qquad \pi_j \in \Gamma(C_i)$$

If the code $\gamma(C_i)$ of a class C_i is represented as a bit vector, then the encoding length can be computed using the binary logarithm $lg(x)$:

$$\mid \gamma(C_i) \mid = lg(\prod_j \pi_j) = \sum_j lg(\pi_j) \qquad with \qquad \pi_j \in \Gamma(C_i) \tag{6}$$

The encoding length of a class is always smaller than those of its descendants:

$$\forall C_a, C_b \in \chi : \quad C_b <: C_a \qquad \Rightarrow \qquad \mid \gamma(C_a) \mid < \mid \gamma(C_b) \mid \tag{7}$$

A class C_b has at least one extra prime number in its factorization compared to its parent or ancestor class C_a, i.e. its personal gene g_B. Since the smallest prime number is 2, the multiplication of the genes in $\Gamma(C_b)$ will at least be 1 bit longer than the encoding of class C_a.

The encoding of a multiple inheritance hierarchy $\chi = \{A, B, C, D, E\}$ is illustrated in Figure 9. The $O1$ label refers to the fact that the classes are defined in *ontology 1*. This label can be ignored for the time being, but it will be needed later on. The encoding of each class is a multiplication of prime numbers. The personal genes of each class are underlined in Figure 9. For example, class E inherits the genes 2 and 7 from its ancestors and is assigned 11 as its personal gene g_E. The encoding $\gamma(E) = \prod_j g_j$ with $g_j \in \Gamma(E)$ becomes $\gamma(E) = 2 * 7 * 11 = 154$. As classes A and D have no ancestors, their encoding $\gamma(A)$ and $\gamma(D)$ is solely based on their personal gene $g_A = 2$ and $g_D = 7$.

Theorem 1 A class $A \in \chi$ is subsumed by a class $B \in \chi$ if and only if the encoding $\gamma(B)$ of class B divides the encoding $\gamma(A)$ of class A.

$$\gamma(A) \subseteq \gamma(B) \quad \Leftrightarrow \quad \gamma(A) \bmod \gamma(B) = 0$$

Proof We will provide a proof for the implication in both directions:

[⇒] We need to prove that: $\gamma(A) \subseteq \gamma(B) \Rightarrow \gamma(A) \bmod \gamma(B) = 0$.
If a class $A \in \chi$ is subsumed by a class $B \in \chi$, then by definition:

$$\gamma(A) \subseteq \gamma(B) \quad \Leftrightarrow \quad A \equiv B \quad \vee \quad A <: B$$

If $A \equiv B$, then by using the definition in (2) we conclude that both classes have the same encoding: $\gamma(A) = \gamma(B)$

$$\gamma(A) = \gamma(B) \ \wedge \ \gamma(A) \bmod \gamma(A) = 0 \quad \Rightarrow \quad \gamma(A) \bmod \gamma(B) = 0$$

If $A <: B$, then class A inherits all genes from class B, or $\Gamma(B) \subseteq \Gamma(A)$. Using the definition in (2):

$$\Gamma(B) \subseteq \Gamma(A) \quad \Rightarrow \quad \gamma(A) = q * \gamma(B)$$

and:

$$q = \prod_j \ g_j \qquad \text{with} \qquad g_j \in \Gamma(A) \backslash \Gamma(B)$$

Corollary, this means that $\gamma(A) \bmod \gamma(B) = 0$.

[⇐] We need to prove that: $\gamma(A) \subseteq \gamma(B) \Leftarrow \gamma(A) \bmod \gamma(B) = 0$.
If the encoding of class B divides the encoding of class A, then:

$$\gamma(A) \bmod \gamma(B) = 0 \quad \Rightarrow \quad \gamma(A) = q * \gamma(B) \qquad \text{with } q \in \mathbb{N}$$

According to (5), the encodings of classes A and B can be defined as follows:

$$\gamma(A) = \prod_k \ g_k \ \text{ with } \ g_k \in \Gamma(A), \quad \gamma(B) = \prod_l \ g_l \ \text{ with } \ g_l \in \Gamma(B)$$

Combining these definitions results in:

$$\gamma(A) = q * \gamma(B) \quad \Rightarrow \quad \prod_k \ g_k = q * \prod_l \ g_l.$$

Each g_l on the right divides the product on the left of the equation. Since all $g \in G$ are pairwise co-prime, there is a k for each l such that $g_k = g_l$, or:

$$\Gamma(B) \subseteq \Gamma(A) \quad \Rightarrow \quad \gamma(A) \subseteq \gamma(B)$$

Theorem 2 A class $A \in \chi$ is subsumed by a class $B \in \chi$ if and only if the gene $g_B = \varphi(B)$ of class B divides the encoding $\gamma(A)$ of class A.

$$\gamma(A) \subseteq \gamma(B) \quad \Leftrightarrow \quad \gamma(A) \bmod g_B = 0$$

Proof We will again provide a proof for the implication in both directions:

[⇒] We need to prove that: $\gamma(A) \subseteq \gamma(B) \Rightarrow \gamma(A) \bmod g_B = 0$.
Theorem 1 states that $\gamma(B)$ divides $\gamma(A)$:

$$\gamma(A) \bmod \gamma(B) = 0 \ \Rightarrow \ \gamma(A) = q{*}\prod_j \ g_j \qquad \text{with } q \in \mathbb{N}, g_j \in \Gamma(B)$$

The definition (2) declares that $g_B \in \Gamma(B)$, and thus:

$\gamma(A) = q * g_B * \prod_j \ g_j \qquad$ with $q \in I\!N, g_j \in \Gamma(B) \backslash \{g_B\}$

This proves that g_B divides $\gamma(A)$.

[$\Leftarrow$] We need to prove that: $\gamma(A) \subseteq \gamma(B) \Leftarrow \gamma(A) \bmod g_B = 0$.
If the gene of class B divides the encoding of class A, then:

$\gamma(A) = q * g_B \qquad$ with $q \in I\!N$

or

$\prod_k \ g_k = q * g_B \qquad$ with $g_k \in \Gamma(A), q \in I\!N$

As all genes, including g_k and g_B, are pairwise co-prime, there is an index k for a gene g_k such that $g_k = g_B$. Using the definition in (2), either $g_B = \varphi(A)$, so that $A = B$, or there is a class $B' \in \chi$ for which $g_B = \varphi(B')$ and $A <: B' \vee A \equiv B'$. Since by definition $g_B = \varphi(B)$ and only equivalent classes may share the same gene, we conclude that $B \equiv B'$, and thus that $\gamma(A) \subseteq \gamma(B)$.

In summary, the function ψ is defined as the multiplication of a set of genes $g_j \in G$, where each g_j is a prime number $\pi_j \in P$. Depending on the kind of optimization, a different prime mapping function $\pi_j = \varphi(C_j)$ will be used.

Theorem 3 Let a hierarchy χ with n classes $C_i \in \chi$ have prime numbers π_i as unique genes $g_i = \varphi(C_i)$, then an encoding exists that is more compact than the n^2 bits for the binary matrix if each $|\Gamma(C_i)|$ is bounded as follows:

$$\forall \ C_i \in \chi : \ |\Gamma(C_i)| \quad < \quad n \ / \ lg(\ n(\ ln \ n + ln \ ln \ n \ ))$$

Proof For a hierarchy χ of n classes $C_i \in \chi$ having genes $\pi_i = \varphi(C_i)$, and encodings $\gamma(C_i) = \prod_j \ \pi_j$, compaction is achieved if either:

$$lg(\prod_j \ \pi_j) \ = \sum_j \ lg(\pi_j) < n \qquad or \qquad \sum_{C_i} \ lg(\prod_j \ \pi_j) = \sum_{C_i} \sum_j \ lg(\pi_j) < n^2 \tag{8}$$

Note that the second condition for compaction for all classes together is a relaxed version of the first compaction condition where the inequality must be true for each single class. Therefore, we will only provide a proof using the first condition. Some observations with respect to prime numbers are in place here. First of all, in mathematics the *prime number function* $\pi(x)$ counts the number of prime numbers less than or equal to a number x. Gauss and Legendre showed that this function can be approximated:

$$\pi(x) \sim \frac{x}{ln(x)} \tag{9}$$

The following formula [32] defines boundaries of $\pi(x)$ for values of $x \geq 17$:

$$\frac{x}{ln(x)} < \pi(x) < 1.25506 * \frac{x}{ln(x)} \tag{10}$$

In [15], Dusart showed that the kth prime p_k is greater than $k(ln \ k + ln \ ln \ k - 1)$ for $k \geq 2$. An upper bound for p_k is $k(ln \ k + ln \ ln \ k)$ with $k \geq 6$.

$$k(ln \ k + ln \ ln \ k - 1) \leq p_k \leq k(ln \ k + ln \ ln \ k) \qquad with \qquad k \geq 6 \tag{11}$$

n	p_n	$2^n/ln(2^n) \approx \pi(2^n)$	$lg(p_n)$	s	$m < n/s$	**max ancestors**
8	19	46	4.24	4.49	1.78	0.78
16	53	5909	5.72	5.92	2.70	1.70
24	89	1008516	6.47	6.70	3.58	2.58
32	131	193635250	7.03	7.24	4.42	3.42
64	311	4.158e+17	8.28	8.48	7.55	6.55
128	503	3.385e+36	8.97	9.69	13.21	12.21
256	1619	6.525e+74	10.66	10.86	23.57	22.57
512	3671	3.778e+151	11.84	12.01	42.63	41.63
1024	8161	2.533e+305	12.99	13.15	77.88	76.88

Table 3: For a given n: the nth prime, the number of primes $\leq 2^n$, the bit size of nth prime, the estimated upper bound for the bit size, and the average maximum number of genes and ancestors

Assume that the encoding of each class $C_i \in \chi$ is defined by the product of $m_i = |\Gamma(C_i)|$ prime numbers $\pi_j \in P$, and that the order of a prime π_j is defined by k_j. This means that (8) can be rewritten as follows:

$$\sum_{j=1}^{m_i} lg(p_{k_j}) < n \qquad or \qquad \sum_{C_i}\sum_{j=1}^{m_i} lg(p_{k_j}) < n^2 \tag{12}$$

After applying the upper boundary on p_{k_j} from (11), the conditions for compaction are refined as follows:

$$\sum_{j=1}^{m_i} lg(k_j(ln\ k_j + ln\ ln\ k_j)) < n \qquad or \qquad \sum_{C_i}\sum_{j=1}^{m_i} lg(k_j(ln\ k_j + ln\ ln\ k_j)) < n^2 \tag{13}$$

Both encoding length optimization conditions can be fulfilled for a given hierarchy χ with n classes $C_i \in \chi$, if m_i and k_j are relatively small. The values m_i for each class $C_i \in \chi$ define how sparse the binary matrix is. For a bit vector of size n, the largest number that can be represented is $2^n - 1$. Using the approximations in (9) and (10), we can estimate the number of primes that can be represented in n bits:

$$\frac{2^n}{ln(2^n)} = \frac{2^n}{n * ln(2)} < \pi(2^n) \tag{14}$$

Given that we need n primes π_i as genes g_i for each class $C_i \in \chi$, then the largest prime $max(\pi_i) = p_n$ will be bounded by the formula in (11). Each prime will require at most $s = lg(n(ln\ n + ln\ ln\ n))$ bits. A lower bound for the number of primes $\pi(x)$ and the maximum size s for representing the largest prime are shown in Table 3 for different values of n. For $|\Gamma(C_i)| = m_i$, the conditions in formula (13) are both fulfilled if:

$$\sum_{j=1}^{m_i} lg(k_j(ln\ k_j + ln\ ln\ k_j)) < n \tag{15}$$

with

$$\sum_{j=1}^{m_i} lg(k_j(ln\ k_j + ln\ ln\ k_j)) \leq \sum_{j=1}^{m_i} s = m_i * s \tag{16}$$

Combining both inequalities, this results in:

$$m_i < n \ / \ s \quad \Leftrightarrow \quad m_i < n \ / \ lg(\ n(\ ln \ n + ln \ ln \ n \ )) \tag{17}$$

Compaction is thus achieved if:

$$\forall \ C_i \in \chi : \ |\Gamma(C_i)| < n \ / \ lg(\ n(\ ln \ n + ln \ ln \ n \ )) \tag{18}$$

This illustrates that for a given n, an upper bound $s = lg(\ n(\ ln \ n + ln \ ln \ n \ ))$ can be computed for the bit size of the nth prime $lg(p_n) \leq s$. This maximum bit size s is also an upper bound for the size of any other prime $\pi_j < p_n$. An encoding for each class $C_i \in \chi$ defined as $\gamma(C_i) = \prod_{j=1}^{m_i} \ \pi_j$ is guaranteed to use less than n bits if $\Gamma(C_i)$ contains less than n/s genes, i.e. $m_i = |\Gamma(C_i)| < n/s$, for encoding C_i. In practice, m_i can be a lot larger because the size of all primes $\pi_j < p_n$ will be smaller than $lg(\pi_j) < lg(p_n) \leq s$ bits.

The above discussion outlines that prime numbers can be used for a compact representation of classes in a hierarchy. The degree of compaction solely depends on the prime number π_i that is assigned to a class C_i, and thus on the mapping function $\pi_i = \varphi(C_i)$. In a following section, we will discuss several heuristics to enable compaction and achieve bit vectors of comparable size to those of the previously described algorithms.

5.2 Conflict-free and fast incremental encoding with prime numbers

As each non-equivalent class $C_i \in \chi$ is assigned a distinct prime number $\pi_i = \varphi(C_i)$ as its personal gene, conflicts for subsumption testing never arise. For encoding a new class C_{n+1} that is incrementally added to a hierarchy χ of n classes $C_{1..n}$, the next available prime number $p_{n+1} = \varphi(C_{n+1})$ is used. As conflicts are of no concern, encoding leaf classes becomes very straightforward. Moreover, there is no need to traverse the hierarchy to collect the genes of the ancestors. Instead, the new code for class C_{n+1} can be computed using the *least common multiple* function $lcm(\{ \ \gamma(C_a), \gamma(C_b), \gamma(C_c), ... \ \})$ of the encoding of its parents:

$$\gamma(C_{n+1}) = p_{n+1} \ * \ lcm(\{ \ \gamma(Z) \ | \ Z \in parents(C_{n+1}) \ \})$$

If the new class C_{n+1} is added into the hierarchy χ as a non-leaf class, then the encoding of all its descendants should be updated by multiplying their encoding with p_{n+1} and the genes of any new ancestors to reflect the inheritance of the descendants from class C_{n+1} and its ancestors. Note that incremental encoding only ensures conflict-free encoding, not the most compact representation.

5.3 Incremental encoding and reuse in ontologies

The basic nature of ontologies is that they are reused as much as possible. Reuse in this perspective means that either an entire ontology is reused that already models a certain domain, or that concepts defined therein are reused in new ontologies. This way, terminology is shared and knowledge is linked. In previous sections we discussed why incremental encoding was important for subsumption under an open world assumption. If new classes in new ontologies inherit from previously defined classes in existing ontologies, then subsumption

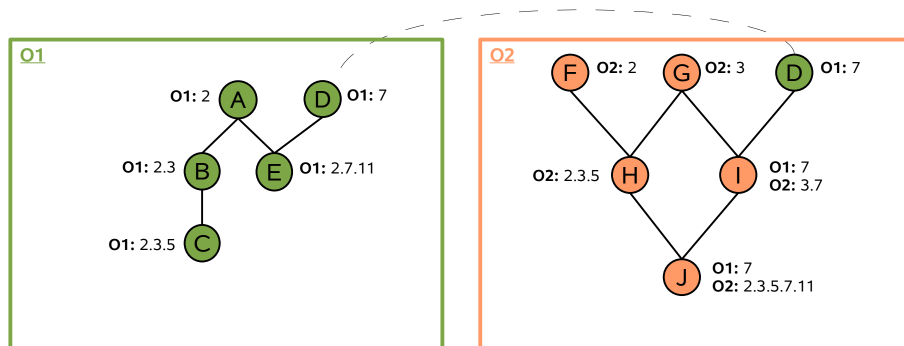

Figure 10: Incremental encoding in multiple ontologies

testing between classes from both ontologies should be handheld efficiently as well. Reuse then also means that it should be possible to carry out the encoding of an ontology once in advance, and that everyone can reuse the encoding for quick subsumption testing.

In a previous section we argued that to distinguish different classes that each class should be assigned a different prime number as personal gene. This statement also holds for classes defined in multiple ontologies. One way to ensure this is to assign a unique prime number to all classes in all ontologies as if they were defined in the same ontology. However, this would lead to very long encodings after multiplication and it would be impossible to maintain encoding consistency when encoding newly defined ontologies in parallel. We therefore opted to label each ontology and the encodings of the classes defined therein. This enables reuse of smaller primes to achieve shorter encodings, while still being able to distinguish classes that were assigned the same prime number. As a result, if classes inherit from ancestor classes defined in different ontologies, then they may receive multiple encodings. This is illustrated in Figure 10 for classes I and J inheriting from classes defined in two ontologies labeled $O1$ and $O2$. In this example the encodings of the classes in ontology $O1$ did not need to be updated.

A single multiplication is required when a new class is introduced as a parent of previously defined classes. This is illustrated in Figure 11 where class K is defined in ontology $O3$ as a parent of class B defined in ontology $O1$. Not only should the encoding of class B make clear that it inherits from class K, but also all the descendants of class B, such as class C, need to be updated. This example only required the encoding of class K in $O3$ to be added to classes B and C. However, a new ontology may also introduce inheritance relationships between previously defined and unrelated classes. This is the case for classes E and H. As according to ontology $O3$ class H not only inherits from K, but also from E, the encodings of H and its descendants need to be updated. Note that descendant class J already inherited from class D, and thus already acquired an encoding for ontology $O1$, but it needs to be updated because it also inherits from class E defined in the same ontology. Of course, as many ontologies may

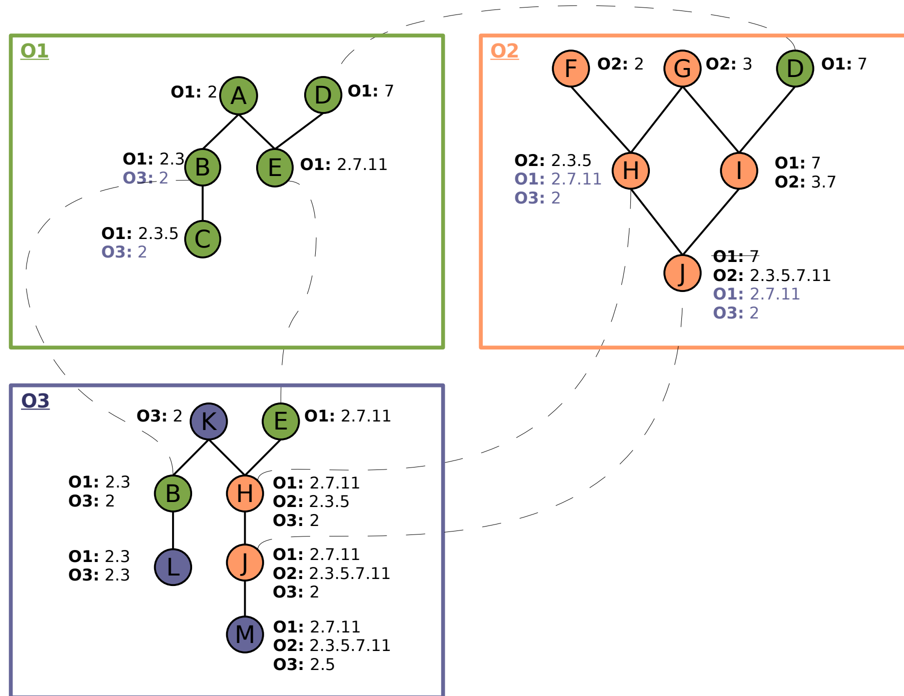

Figure 11: Re-encoding descendants in multiple ontologies

link to the same ontologies, continuously updating the encoding of a popular ontology is problematic. Therefore, any new encodings for previously defined classes are stored in the encoding of the ontology that triggers these updates. This way, an ontology can reuse the smaller prime numbers in order to keep the encoding length to a minimum. The labels are derived from the unique URI of the ontology in order to distinguish different ontologies.

5.4 Incremental encoding and equivalent classes

When incrementally encoding classes in a new ontology, one has to ensure that conflicting relationships are avoided. For example, assume as in Figure 12 that a class J is defined in ontology $O2$ to inherit from a class I. This means that:

$$J <: I \quad \Rightarrow \quad \gamma(J) \subseteq \gamma(I)$$

After combining the multiple inheritance hierarchies of the two ontologies $O3$ and $O4$, class I indirectly inherits from class J through the class M defined in ontology $O3$ and reused in ontology $O4$:

$$I <: J \quad \Rightarrow \quad \gamma(I) \subseteq \gamma(J)$$

This cross-inheritance relationship is only possible if classes I and J are equivalent, i.e. if all instances of class I are also instances of class J at all times, and vice versa:

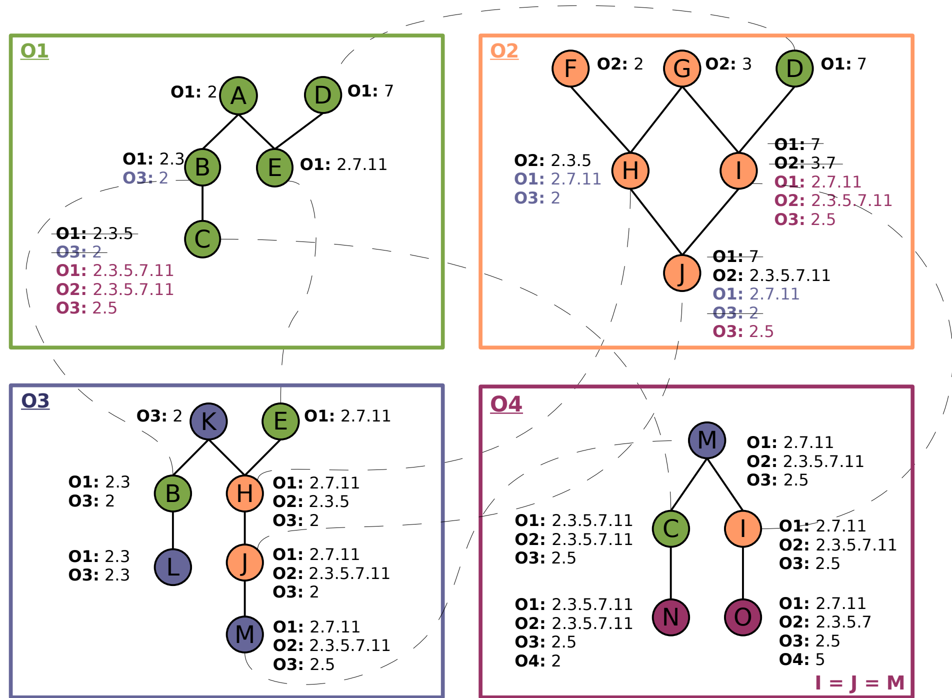

Figure 12: Incremental encoding and equivalent classes

$$\gamma(J) \subseteq \gamma(I) \quad \wedge \quad \gamma(I) \subseteq \gamma(J) \quad \Leftrightarrow \quad I \equiv J$$

This equivalence is ensured by the fact that the encoding of a class can always be divided by the least common multiple of the encodings of its parents. As a result, any class in the combined multiple inheritance hierarchy in between classes I and J is also equivalent to these equivalent classes. This is the case for class M. Note that the examples given in this and in the previous section are artificial and may not occur that often in real world ontologies. It however illustrates that the prime-based encoding is able to cope with these issues.

6 Optimizing encoding and subsumption testing

In the previous sections we have explained the basic principles of the prime-based encoding of multiple inheritance hierarchies, and how a simple division is used for efficient subsumption testing. The major benefits of the prime-based encoding technique is that each class acquires a new personal gene, so that conflicting codes as those that arose in the related work can be avoided. We proved that under certain conditions an encoding exists that is more compact than the binary matrix representation. In this section, we will present several heuristics for the mapping function $\pi_i = \varphi(C_i)$ in order to achieve compaction results that are comparable to those of the subtyping algorithms discussed in

section 4. These heuristics will either try to minimize the total encoding length for all the bit vectors in the hierarchy, or to minimize the size of largest bit vector representation of a single class. Later on we will discuss how to optimize the efficiency of the subsumption test itself.

6.1 Optimizing the encoding length of the representation

First of all, the order of assigning prime numbers in the example of Figure 9 was more or less determined randomly. Not only does the assignment of a prime number to a class affect the encoding length of the class itself, but also all those of its descendants. By intelligently selecting a prime number for each class, a more compact representation is likely achievable. The encoding length of a bit vector representation can be optimized in two ways. Either the total length of the encodings of all classes is reduced to a minimum, or the longest bit vector representation among all classes is minimized in size.

In the following encoding schemes, we assume that the prime number selection for each new class is done in order, i.e. we start with assigning the first prime number $p_1 = 2$ to the best feasible class according a specific heuristic and continue with the next prime number $p_2 = 3$ for the next class, and so on. A heuristic can thus be defined as a function that maps a prime number p_i with $i = 1..n$ to one of the n classes $C_i \in \chi$ such that $C_i = \varphi^{-1}(p_i)$. Note that no matter what heuristic is used, it is always better to first encode all the ancestors of a class before encoding the class itself.

Theorem 4 Let a hierarchy χ have n classes $C_i \in \chi$ and let n prime numbers p_i with $i = 1..n$ be assigned to each class $C_i = \varphi^{-1}(p_i)$ in order. The encoding length of a class C_i is the smallest if it is encoded before its descendants.

Proof Assume that a class $A \in \chi$ is an ancestor of a class $B \in \chi$, then according to the definitions in (2) and (5):

$$\begin{array}{lll} \gamma(A) = \pi_a * \prod_j \pi_j & \text{with} & \pi_j \in \Gamma(A)\backslash\{\pi_a\} \\ \gamma(B) = \pi_b * \pi_a * \prod_j \pi_j & \text{with} & \pi_j \in \Gamma(B)\backslash\{\pi_b, \pi_a\} \end{array}$$

If the genes π_a and π_b of classes A and B would be exchanged, then only the encoding of class A changes. Using (2), the encoding length of a class $C \in \chi$ is only affected if:

$$C <: A \qquad \vee \qquad C <: B \qquad \text{with} \qquad B <: A$$

Under these conditions class C is always a descendant of class A. The remaining possibilities are:

1. **C** $<:$ **B**: Definition (2) declares that $\Gamma(C)$ contains both π_a and π_b. The encoding length $lg(\gamma(C))$ of class C therefore remains unchanged.

2. **C** $\not<:$ **B**: The set $\Gamma(C)$ only contains gene π_a, but not gene π_b. The encoding of class C is smaller if $\pi_a < \pi_b$:

$$\gamma(C)|_{\pi_a<\pi_b} < \gamma(C)|_{\pi_a>\pi_b} \tag{19}$$

Hierarchy	Classes	Depth	Max Parents	Max Children	Max Ancestors
SUMO	630	16	3	15	19
Wine&Food	133	9	4	13	14
Pizza	99	9	5	23	12
Gene Ont.	20945	16	5	1723	54
Java 1.30	5438	10	14	1128	19
OpenCyc	25565	32	19	366	89

Table 4: Several multiple inheritance hierarchies and their characteristics

If a class C that inherits from class A but not from class B does not exist, then at least for $\pi_a < \pi_b$ the encoding $\gamma(A)$ of class A is smaller than for $\pi_a > \pi_b$.

$$\gamma(A)|_{\pi_a<\pi_b} < \gamma(A)|_{\pi_a>\pi_b} \tag{20}$$

Note that even for two encodings $\gamma(A_1) < \gamma(A_2)$ of a class A, that their bit vector representations may be equally long. Based on the outcome in (19) and (20), we conclude that the encoding length of a class is at least the same if not smaller than for any other encoding order of the class and its descendants.

This optimization is taken into account in most of the heuristics described below. Later on, we will show how the order of encoding is also exploited for fast subsumption testing. We will now describe several heuristics to optimize the encoding length and illustrate their efficiency by means of several ontologies and multiple inheritance hierarchies:

- **SUMO**: The SUMO ontology [27] is the *Suggested Upper Merged Ontology* developed within the IEEE Standard Upper Ontology Working Group for the purpose of developing a standard ontology that will promote data interoperability, information search and retrieval, automated inferencing, and natural language processing.
- **Wine & Food**: These two well-known ontologies, initially developed by McGuinness [28], are often used to explain the basic features and some of the more advanced concepts of the OWL DL ontology language [35].
- **Pizza**: This example ontology is used in the Protege-OWL tutorial and contributed by the CO-ODE Project [14].
- **Gene Ontology**: The Gene Ontology project [18] is a joint effort to create a vocabulary of genes from any organism [36] and to describe associated gene products in order to facilitate uniform queries across them.
- **Java 1.30 Types**: This hierarchy is not an ontology, but part of a benchmark [45] suite used to compare the compactness of several subtyping algorithms.
- **OpenCyc**: This is one of the largest upper ontologies [10] used as a general knowledge base of everyday common sense with the aim to enable applications to perform human like reasoning. In our experiments, we use version v0.7.8b of the OWL implementation.

An overview of several properties of these multiple inheritance hierarchies is given in Table 4. This set of hierarchies covers various ranges of number of classes, depth, maximum number of parents, children and ancestors. Using

these hierarchies we will compare the compactness of the encodings achieved by several heuristics. Note that an encoding that is more compact than the binary matrix representation is guaranteed according to the maximum ancestors results in Table 3. In the following sections we discuss several heuristics and provide a simplified implementation in pseudo-code.

6.1.1 Heuristic 1: Semi-random top-down order

This heuristic is very straightforward: the algorithm starts with a list containing all the root classes and assigns a prime number to the first class in the list. It removes this class from the list and appends its unencoded children to the list. The algorithm then repeats this process until the list is empty. This heuristic is called semi-random because the children of a class are encoded in a more or less random order, but the encoding itself is top-down. As such, the encoding is a bit more intelligent than a completely random encoding where prime numbers are randomly assigned to all classes in the hierarchy. Note that this heuristic does not comply with Theorem 4 as the first root class may have a child that occurs as a grand child of the second root class. Therefore, it is possible that the grand child is assigned a prime number before its parent is assigned one. A pseudo-code implementation of this hierarchical encoding algorithm is shown below in full. The hierarchy containing the classes C serves as input. The output of this algorithm is the encoding $\gamma(C)$ for each class C:

```
EncodeHierarchy(in: hierarchy, out: gamma) {
    n = SizeOf(hierarchy)
    primeTable[1..n] = ComputePrimes(n)
    i = 0
    classList = Roots(hierarchy)
    while SizeOf(classList) > 0 do {
        // Determine the next best class (Heuristic 1)
        bestClass = First(classList)

        // Assign the next gene to the class and its descendants
        i = i + 1
        AssignPersonalGene(bestClass, primeTable[i])
        AddInheritedGene(Descendants(bestClass), primeTable[i])

        // Remove the class and add its children to the list
        RemoveClass(classList, bestClass)
        AddClass(classList, NoPersonalGene(Children(bestClass)))
    }
    // Compute the encoding for all classes
    for each C in hierarchy do
        gamma[C] = MultiplyAllGenes(C)
}
```

6.1.2 Heuristic 2: Class with most descendants

This heuristic sorts the classes in the list according to their number of descendants. The one with the most descendants is assigned a prime number first. This heuristic is based on the principle that all the descendants of a class profit from a shorter shared prime number assigned to their common ancestor. This

heuristic respects Theorem 4 as an unassigned class will always have less descendants than one of its unassigned ancestors. The pseudo-code of this heuristic differs from the first heuristic with respect to the next best class selection:

```
// Determine the next best class (Heuristic 2)
bestClass = First(classList)
bestCount = SizeOf(Descendants(bestClass))
for each C in classList do {
    count = SizeOf(Descendants(C))
    if count > bestCount then {
        bestClass = C
        bestCount = count
    }
}
```

6.1.3 Heuristic 3: Leaf class with most ancestors

This heuristic starts by determining the leaf classes in the hierarchy, and ordering them according to their number of ancestors. This is the corresponding pseudo-code of the encoding algorithm:

```
EncodeHierarchy(in: hierarchy, out: gamma) {
    n = SizeOf(hierarchy)
    primeTable[1..n] = ComputePrimes(n)
    i = 0
    // Sort the leaves according to their ancestors (Heuristic 3)
    sortedLeafList = SortMostAncestors(Leaves(hierarchy))
    for each L in sortedLeafList do {
        ancestorList = NoPersonalGene(Ancestors(L))
        while SizeOf(ancestorList) > 0 do {
            // Determine the next best ancestor using Heuristic 2
            bestAncestor = First(ancestorList)
            bestCount = SizeOf(Descendants(bestAncestor))
            for each C in ancestorList do {
                count = SizeOf(Descendants(C))
                if count > bestCount then {
                    bestAncestor = C
                    bestCount = count
                }
            }

            // Assign the next gene to the ancestor and its descendants
            i = i + 1
            AssignPersonalGene(bestAncestor, primeTable[i])
            AddInheritedGene(Descendants(bestAncestor), primeTable[i])

            // Remove the ancestor from the list
            RemoveClass(ancestorList, bestAncestor)
        }

        // Assign the next gene to the leaf
        i = i + 1
        AssignPersonalGene(L, primeTable[i])
    }
```

```
    // Compute the encoding for all classes
    for each C in hierarchy do
        gamma[C] = MultiplyAllGenes(C)
}
```

A class with many ancestors is likely to have a long encoding due to the multiplication of all its inherited genes. We therefore sort the leaf classes according to a decreasing number of ancestors. The ancestors of the first leaf class in the list are assigned a personal gene according to heuristic 2, i.e. sorting the ancestors according to their number of descendants. The algorithm then assigns a prime number to the leaf class and continues to process the next leaf in the sorted list. For any next leaf class, it is possible that some of its ancestors have already received a prime number during a previous personal gene assignment. For all such ancestors with a personal gene, we know that their own ancestors have also received a personal gene in order to comply with Theorem 4.

6.1.4 Heuristic 4: Leaf class with the largest minimum code

This is the most complex heuristic of all of them. It tries to keep the size of the longest code as short as possible and complies with Theorem 4. This longest code will be achieved in one of the leaf classes of the hierarchy. The algorithm starts with a list containing all root classes. It then continuously applies the following steps:

1. Determine for each leaf class what its *minimum code* (MC) is given the current partial gene assignment. The MC of a leaf class would be its encoding if all its ancestors without a personal gene and the leaf itself were immediately assigned a prime number. Assume there are n such ancestors, then the MC of a leaf can be computed by multiplying its currently inherited genes with the next available $(n + 1)$ prime numbers.

2. Select the leaf class with the largest MC. For all the classes in the list that have a leaf with the same largest MC, assign a prime number as a personal gene to the class that has the most descendants.

3. Remove the class from the list and add each child to the list as long as all the child's ancestors have been assigned a personal gene. This constraint on the child is required to comply with Theorem 4.

Steps 1 and 2 are implemented in the *MinimumCode* method shown below, steps 3 and 4 in the following methods.

```
MinimumCode(in: class, lastPrimeIndex, out: code) {
    // Count the class' ancestors without a personal gene
    n = SizeOf(NoPersonalGene(Ancestors(class)))

    // Multiply the inherited genes received from the other ancestors
    code = MultiplyInheritedGenes(class)

    // Compute the code if the class and its ancestors would be assigned
    // a personal gene
    for j=1 to (n+1) do
        code = code * primeTable[lastPrimeIndex + j]
}
```

```
LargestMinimumCode(in: classList, lastPrimeIndex, out: bestClass) {
    bestClass = First(classList)
    largestMinimumCode = MinimumCode(bestClass, lastPrimeIndex)
    for each C in classList do {
        minimumCode = MinimumCode(C, lastPrimeIndex)
        if minimumCode > largestMinimumCode then {
            bestClass = C
            largestMinimumCode = minimumCode
        }
    }
}

EncodeHierarchy(in: hierarchy, out: gamma) {
    n = SizeOf(hierarchy)
    primeTable[1..n] = ComputePrimes(n)
    i = 0
    classList = Roots(hierarchy)
    leafList = Leaves(hierarchy)
    while SizeOf(classList) > 0 do {
        // Determine the next best class (Heuristic 4)
        bestClass = First(classList)
        bestCount = SizeOf(Descendants(bestClass))
        bestCode = LargestMinimumCode((Leaves(bestClass)), i)
        for each C in classList do {
            code = LargestMinimumCode((Leaves(C)), i)
            if code > bestCode then {
                bestClass = C
                bestCount = SizeOf(Descendants(bestClass))
                bestCode = code
            } else if code = bestCode then {
                count = SizeOf(Descendants(C))
                if count > bestCount then {
                    bestClass = C
                    bestCount = count
                }
            }
        }

        // Assign the next gene to the class and its descendants
        i = i + 1
        AssignPersonalGene(bestClass, primeTable[i])
        AddInheritedGene(Descendants(bestClass), primeTable[i])

        // Remove the class and add its children to the list
        RemoveClass(classList, bestClass)
        for each C in Children(bestClass) do {
            if AllParentsHavePersonalGene(C) then
                AddClass(classList, C)
        }
    }
    // Compute the encoding for all classes
    for each C in hierarchy do
        gamma[C] = MultiplyAllGenes(C)
}
```

Hierarchy	Heuristic 1		Heuristic 2		Heuristic 3		Heuristic 4	
	max	total	max	total	max	total	max	total
SUMO	175	35569	113	26338	94	31665	83	29995
Wine&Food	99	4206	66	2946	57	3298	53	3237
Pizza	64	3146	50	2224	49	2565	40	2382
Gene Ont.	863	2997019	478	1710174	390	2150914	358	2052069
Java 1.30	259	242121	185	164206	134	209708	112	201802
OpenCyc	1266	12038123	821	6953122	905	8242696	681	7506889

Table 5: Encoding length in bits for the different heuristics and hierarchies

These 4 steps are repeated until the list is empty and all the classes have been assigned a prime number as their personal gene.

The results of these heuristics for the multiple inheritance hierarchies mentioned earlier are shown in Table 5. For each heuristic the table shows the size of the longest bit vector representation and the total bit size of all encodings together. Heuristic 1 showed to be the least intelligent, as was to be expected. Since heuristic 2 took into account the descendants that would benefit from a short prime code, its clear that this heuristic would result in the overall smallest encoding. The fourth heuristic tried to minimize the largest code in the hierarchy. As a result, this heuristic achieves the smallest maximum bit vector size.

6.2 Optimizing the subsumption test

In theorem 2 we proved that a class $A \in \chi$ is subsumed by a class $B \in \chi$ if and only if the gene $g_B = \varphi(B)$ of class B divides the encoding $\gamma(A)$ of class A. Given the sizes of the bit vectors in Table 5, it is obvious that a machine word length of 32 or 64 bits will not suffice. Therefore, mathematical libraries with support for arbitrary precision are required. However, using such libraries always brings along some overhead that can be avoided. Our current subsumption testing algorithm is implemented in Java and in C. We therefore investigated the division algorithms of the `java.math.BigInteger` class in several Java classpath implementations, the GNU Multiple-Precision Library, as well as several others. We will refer to numbers of more that 32 or 64 bits as *big integers*. Here are some remarks with respect to the division operation for big integers and possible optimizations that can achieved for our specific setup:

- Most libraries only provide an interface for the division operation that expects two big integers as parameters. For our purpose, it is very likely that g_B fits within a machine word.
- Each of the implementations takes into account the sign of both integers. As both our numbers are positive, the logic for determining the sign of the result can be left out.
- We are only interested in the remainder after the division, not in the quotient. However, many libraries make use of an internal function that computes the quotient and the remainder at the same time.
- In fact, in our test setup we only need to know if the division succeeds or not. No big integer variable for storing the remainder should be allocated either as it will be discarded anyway.

- Given the fact that the real value of the quotient and remainder is not important, several optimizations are possible that avoid most the division arithmetic. These optimizations will be discussed below.

As a result of these observations, we have decided to implement a fast method ourselves to check whether a big integer can be divided by an integer of less than 32 bits. Given that the largest prime number of 31 bits is larger than 193635250 (see Table 3), we do not assume an ontology with that many classes will occur in the near future soon. By the time this number is reached, 64-bit computing will be mainstream already. Our implementation of big numbers makes use of an array of 32-bit integers that dynamically grows when needed. The representation is the same as a 32 bit integer modeled as an array of 4 bytes. Some other optimizations to decide whether a class $A \in \chi$ with encoding $\gamma(A)$ and prime number π_A as personal gene subsumes a class $B \in \chi$ with encoding $\gamma(B)$ and prime number π_B are discussed below.

6.2.1 The bit vector length of both classes

A class A will never subsume a class B if the size of the bit vector representation of its encoding is larger than the size of the bit vector of class B. This results from the fact that a descendant of class A always inherits all genes from A and has its own personal gene. It is thus larger by definition:

$$|\gamma(A)| > |\gamma(B)| \qquad \Rightarrow \qquad \gamma(B) \not\subseteq \gamma(A)$$

Since two non-equivalent classes where one inherits from the other always have one gene not in common, we know that the size of their bit vector representation will always differ with at least one bit as the smallest prime number is 2. Therefore, if the sizes of the bit vectors are the same then the personal genes will determine whether the classes are equivalent or not:

$$\begin{array}{lllll} |\gamma(A)| = |\gamma(B)| & \wedge & \pi_A = \pi_B & \Rightarrow & A \equiv B \\ |\gamma(A)| = |\gamma(B)| & \wedge & \pi_A \neq \pi_B & \Rightarrow & A \not\equiv B \end{array}$$

Of course, equivalent classes subsume one another.

6.2.2 The personal prime number of both classes

If the encoding of the hierarchy complies with Theorem 4, then subsumption for class A of class B can be ruled out very quickly if:

$$\pi_A > \pi_B \qquad \Rightarrow \qquad \gamma(B) \not\subseteq \gamma(A)$$

The equality of personal genes could be used to determine the equivalence of classes A and B, but this test is very unlikely to succeed and is therefore not used except in the test above.

6.2.3 Principal prime number analysis

Not all divisions can be avoided with the above optimizations. This means that a division operation with a big integer will have to be carried out. As a big integer may be several hundred bits long, the implicit computation of the remainder may be a lot slower than a simple binary OR or AND operation

of two 32 or 64-bit machine words, as used in the constant time tests of the previously described subtype testing algorithms.

Luckily, a similar technique can be used for fast subsumption testing between classes. For any prime number $\pi \in P$, a class A will not subsume a class B if a prime number $\pi \in P$ divides the encoding of class A but not the one of class B:

$$\forall \pi \in P : \quad \gamma(A) \bmod \pi = 0 \land \gamma(B) \bmod \pi \neq 0 \quad \Rightarrow \quad \gamma(B) \not\subseteq \gamma(A)$$

If for k prime numbers each class describes in k bits whether each of these k prime numbers occurs in its factorization, then some expensive divisions can be avoided. For two classes A and B, k of the above tests can be executed in parallel with a simple binary AND operation on their two size k bit vectors. Of course, the maximize the benefit, the prime numbers need to be well chosen. For example, a prime number that is inherited by all descending classes will improve nothing. We call these k well chosen prime numbers the k *principal prime numbers* (PPN) of a specific hierarchy encoding.

However, as the size this bit vector is limited to the best k prime numbers, this test may lead to false positives if all the prime numbers that divide the encoding of class A also divide the encoding of class B. Therefore, if they cannot be used to determine that $\gamma(B) \not\subseteq \gamma(A)$, then a division needs to be carried out to compute the correct subsumption result. The k principal prime numbers are determined as follows:

1. Initialize all n entries of a scoreboard S to 0. There is an entry for each of the n prime numbers used in the encoding of the hierarchy χ.

   $$S[p_i] = 0 \qquad \text{with} \qquad i = 1..n$$

2. Test each pair of classes $A, B \in \chi$ if $\gamma(B) \subseteq \gamma(A)$.

   (a) If $\gamma(B) \subseteq \gamma(A)$, continue with the next pair in step 2. This means that a division needs to be carried out for non-equivalent classes to be sure that one subsumes the other.

   (b) If $\gamma(B) \not\subseteq \gamma(A)$ can be successfully tested with either the bit vector size, the personal gene, or a previous principal prime number, then continue with the next pair in step 2.

   (c) For each prime number p_i in the factorization of $\gamma(A)$ that is not present in the factorization of $\gamma(B)$ increase the score of p_i:

   $$S[p_i] = S[p_i] + 1$$

3. The best principal prime number is the prime number p_i with the highest score, i.e. for which $S[p_i]$ is maximal.

4. Go back to step 1 as long as $S[p_i] > 0$ and less than k principal prime numbers have been computed.

Without taking into consideration the bit vector size and personal prime number optimizations, finding the best principal prime number is equivalent to finding the prime number p_i that divides the remaining N classes in two groups: those that are divided (T) by p_i and those that are not divided (F) by p_i. The score of the prime number p_i is then:

$$S[p_i] = T \ * \ F = T * (N - T) = T * N - T^2$$

Hierarchy	Tests	$> \|\gamma(\mathbf{X})\|$	$\pi_{\mathbf{X}}$	PPN	$= \|\gamma(\mathbf{X})\|$	Remaining
SUMO	396900	194098	129194	67013	1419	5176 (=1.3%)
Wine&Food	18769	8915	5080	3937	204	633 (=3.4%)
Pizza	9801	4594	2780	1754	177	496 (=5.1%)
Gene Ont.	438693025	216291499	177244558	40463574	1756726	2936668 (=0.7%)
Java 1.30	29571844	14112491	11930735	1551254	545471	1431893 (=4.8%)
OpenCyc	653569225	325847041	281038733	41834160	303875	4545416 (=0.7%)

Table 6: Optimizing the division operation for subsumption testing

Hierarchy	BigInteger	W/O PPN	W/ PPN	GCJ 4.1	Native C
SUMO	273 ms	41 ms	31 ms	31 ms	4 ms
Wine&Food	38 ms	11 ms	11 ms	2 ms	0 ms
Pizza	27 ms	6 ms	4 ms	1 ms	0 ms
Gene Ont.	381483 ms	37599 ms	25529 ms	35585 ms	4696 ms
Java 1.30	15185 ms	1645 ms	1511 ms	2278 ms	354 ms
OpenCyc	1251725 ms	77962 ms	41419 ms	52271 ms	7628 ms

Table 7: Performance of several subsumption test implementations

To maximize this score, the T partial derivative should be 0:

$$\frac{\partial S[p_i]}{\partial T} = 0 \quad \Leftrightarrow \quad N - 2T = 0 \quad \Leftrightarrow \quad T = N/2 \tag{21}$$

This shows that the best results are achieved if each new principal prime number splits the remaining N classes into two groups of equal size.

Note also that this optimization technique is very similar to the concept of Bloom filters [4]. A Bloom filter is a space-efficient data structure to test whether an element is a member of a set. It uses a bit vector of l bits long and k hash functions that map a key value to one of the l bit positions. Each element in the set is passed through the k hash functions, and the corresponding bit positions are set to 1. Later on, when testing whether an element is in the set, the element is also passed through the k hash functions. If one of the resulting bit positions is 0, then the element is not in the set. Otherwise, the bit position may have been set to 1 by the element itself or by another element. Similar to our technique, the Bloom filter allows false positives but no false negatives. The difference, however, is that it only uses one bit vector of size l as data structure, whereas our technique uses many small bit vectors of size k. The k principle prime numbers defined earlier correspond to the k hashing functions. The more principal prime numbers or hash functions are used, the better the result.

In Table 6 we show the result of an experiment, where each pair in the multiple inheritance hierarchy encoded according to heuristic 4 was tested for subsumption. This means that for a hierarchy with n classes, n^2 subsumption tests were conducted. For a multi-precision library this would mean n^2 divisions of big integers. The results show how many divisions were avoided by each optimization. In the experiment we allowed up to k=64 principal prime numbers, but in some cases less than 64 prime numbers were required as all false positives were already eliminated. As you can see, the number of actual divisions that have been avoided is considerably large.

In Table 7 we illustrate the performance of several subsumption test implementations. The test results were obtained on a Pentium Duo Core 1.83 GHz machine and show the time in milliseconds that was needed to execute all pairwise subsumption tests. In the first column the `java.math.BigInteger`

Ontology	Bin. Matrix	Caseau	Krall	CHNR	Prime Max	Prime Avg
SUMO	630	48	30	45	83	42
Wine&Food	133	39	33	?	53	23
Pizza	99	40	37	28	40	23
Gene Ont.	20945	2155	151	?	361	82
Java 1.30	5438	1568	68	?	112	31
OpenCyc	25565	1420	350	?	681	272

Table 8: Comparison of encoding length of a single class in bits

implementation of a JDK 1.5 virtual machine was used. These are compared against are own optimized Java division implementation, with and without the use of the principal prime numbers. An important remark is that for the 3rd column, at least 50% of the time was spent in code for iterating all pairs in the double loop. This was tested by going through the double loop without actually carrying out the subsumption test. The last but one column shows the performance of the same Java code compiled to a native binary using the GNU compiler for Java. The last column shows the performance of the same algorithm implemented in the C language. Our division based algorithm outperforms the existing implementations. In the following section we will compare our algorithm with other encoding techniques and ontology reasoners.

One last remark is in place here. The experiments conducted here were only meant to benchmark the subsumption testing algorithm. By no means do we claim that pairwise testing of the subsumption relationship for all classes is useful in any way.

7 Experimental evaluation and validation

In the previous sections we have illustrated some heuristics for a compact representation of a hierarchy encoding and for fast subsumption testing. We have conducted experiments to prove their efficiency. We will now compare them with other encoding techniques and ontology reasoners.

7.1 Subtyping algorithms

In section 4 we reviewed several encoding algorithms that are used for efficient constant time subtype testing. Although this is not the focus of the proposed subsumption algorithm, we will show that our encoding schemes fare well next to the other techniques.
Table 8 provides an overview of the encoding lengths achieved by various algorithms. The results show the encoding length for a class in the hierarchy, expressed in bits. For the binary matrix method this is equal to the number of classes in the hierarchy. For our prime-based hierarchy the results for *Prime Max* were copied from the encoding in Table 5 using heuristic 4, and those for *Prime Avg* were obtained by dividing the total encoding length of heuristic 2 with the number of classes. The other results were achieved by using programs freely made available on the Internet [4] [5], or by using our own implementation. The program we used for CHNR was not able to encode other structures than

[4]http://perso.ens-lyon.fr/eric.thierry/bitvector.html
[5]http://www.complang.tuwien.ac.at/andi/typecheck/

Ontology	OWL Pellet		Racer		Prime		Native Prime	
	time	mem	time	mem	time	mem	time	mem
SUMO	4659 ms	9M	26 s	19M	31 ms	181 K	4 ms	59K
Wine&Food	278 ms	26M	1400 ms	21M	11 ms	139 K	0 ms	14K
Pizza	181 ms	6M	660 ms	19M	4 ms	135 K	0 ms	10K
Gene Ont.	265 min	220M	566 min	117M	26 sec	2095 K	5 sec	1836K
OpenCyc	538 hrs	590M	153 hrs	295M	42 sec	3129 K	8 sec	3048K

Table 9: Time and memory consumption for pairwise subsumption testing

trees. Therefore, we were not able to provide complete results for this encoding scheme. Note however, as some techniques use graph coloring, it is expected that better results can be achieved if better heuristics and more computing time is permitted. However, since graph coloring is NP-hard and ontologies can be very large, the encoding time of a hierarchy may become an issue.

As it is not clear if some of the algorithms would gain much from a variable length representation of the bit vector, we compare our prime-based approach for both scenarios using 2 different encoding schemes. If each bit vector has the same size, then the *Prime Max* column shows our encoding is sometimes better, sometimes worse. If variable length encoding can be used, then the results in the *Prime Avg* column show that the encoding length can be severely reduced.

Many other subtyping algorithms exist, but due to lack of implementations, we were not able to compare our encoding results with these algorithms. Nor did we test the speed of subtype testing. As binary operators are commonly used on two bit vectors to determine a subsumption relationship between two classes, we assume that the largest impact factor on such results would be the length of the bit vector, especially if this vector is longer than the length of a machine word, nowadays 32 or 64 bits. Nonetheless, since our subsumption testing mechanism was mainly focused on the encoding of concepts in ontologies, we will carry out some performance tests with ontology reasoners in the following section.

7.2 Subsumption with ontology reasoners

We also compared the subsumption test efficiency with two popular OWL reasoners: Racer [21] and OWL Pellet [30]. Both performance and memory consumption were measured. The results are shown in Table 9. The time is measured in either milliseconds, seconds, minutes or hours. Memory consumption is expressed in kilobytes or megabytes. It is clear that both our Java version *Primes* and our native implementation in the C language *Native Prime*, beat the ontology reasoners. Note that the memory consumption of the Java version only takes into account the used heap space of the *Prime* application, not the memory requirements for the Java virtual machine itself. At least for the RACER engine, we could find a reason for its bad performance. Subsumption queries had to be sent to the server port of the standalone RACER engine, and this may have taken some time. However, we are not sure if the communication part is the only one to blame. We also tried the FaCT++ [38] reasoner, but we were not able to reliably measure the time for the subsumption testing due to the use of the DIG description logic reasoner interface. The results are not included because we are almost certain after several experiments that the time required to process the XML-based DIG format outweighs the time consumed for the subsumption test itself.

7.3 Modeling other ontological features with prime numbers

In the previous sections we have mainly discussed the subsumption relationship for classes. Ontology languages such as OWL [26] provide many more logical constructs to define classes, other than subclassing from previously defined classes. In this section we will cover which features of the OWL language can be modeled using prime numbers. However, by no means do we intend to implement a whole description logic language and reasoner based on prime number arithmetic, as some of the OWL features will likely cause a severe performance penalty on the simplicity of some of our inference operations. The extensions discussed below are not all implemented but serve as a proof-of-concept:

- **Property**: The OWL language supports *DatatypeProperty* and *ObjectProperty* properties. The first model attributes of the primitive data type kind, such as strings and integers, whereas the latter define binary relationships between instances of classes. Properties can be encoded in the same way as classes. The least common multiple of the genes of a class' properties can be used to encode whether the class has a specific (subsuming) property defined.
- **Domain and Range**: The domain specifies a set of classes which have the property defined, and the range specifies a set of classes to restrict the type of possible values. A set of classes is encoded by multiplying their personal genes. For a set of classes, either the intersection, the complement or the union of the classes is possible. See a following entry how membership of a class to the domain or the range of a property is tested.
- **subPropertyOf**: Property inheritance is possible with the *subPropertyOf* construct. The same encoding scheme for subtyping as previously outlined for classes can be reused to test for property subsumption.
- **equivalentProperty**: The encoding is similar to the encoding of equivalent classes. The same genes and encodings are reused for equivalent properties.
- **inverseOf**: A property p is the inverse of another property if it has, aside from its personal gene g_p, a second personal gene $-g_q$. Modeling this feature was only possible using a second gene, as otherwise inheriting from two properties with negative genes causes ambiguity.
- **unionOf, complementOf and intersectionOf**: These features are modeled by multiplying the personal genes of the classes involved. For *unionOf*, membership testing of a class is carried out by checking if the greatest common divisor of the class' encoding and the multiplied genes differs from 1. For *intersectionOf*, the divisibility by the multiplied genes is tested. For *complementOf*, the original test is negated.
- **Individual**: Support for instances of classes is rather limited. Currently each individual is only tagged with the gene of its class. This tag combined with class subsumption is used for instance checking.

All the other features that involve individuals, such as property restrictions (*allValuesFrom, someValuesFrom*), (in)equality of individuals (*sameAs, differentFrom, AllDifferent*), property characteristics (*TransitiveProperty, SymmetricProperty, FunctionalProperty, InverseFunctionalProperty*), cardinality (*minCardinality, maxCardinality, cardinality*) and other concepts (*oneOf, hasValue, disjointWith*) are not (yet) supported.

8 Conclusions and future work

Multiple inheritance hierarchies are frequently used in many research domains, mainly for matching or querying purposes. The major issue is to compute subsumption relationships quickly. We have discussed related work in the programming languages domain where subtyping is a thoroughly investigated research topic. We discussed the issues at hand for encoding ontologies under an open world view. We have proposed an encoding scheme that overcomes these issues. The scheme is based the use of prime numbers and their characteristics, and is able to encode a multiple inheritance hierarchy efficiently, both in terms of a compact representation and providing a fast subsumption testing algorithm. Our encoding algorithm also offers an interesting alternative for the class subtyping encoding techniques, as it yields a new way of compaction without the need for changing old conflicting codes during incremental encoding.

The results of our algorithms described in the paper speak for themselves. If one is only interested in subsumption testing, as described in the use cases in section 2, it is clear that better approaches exist compared to ontology reasoners. The latter proved to have a large overhead with respect to memory consumption and their probably not so optimized implementations for subsumption testing. One should note that it would be unfair to compare a power horse like a DL reasoner with a vastly optimized and dedicated algorithm for a single problem, such as subsumption. DL reasoners prove their usefulness when handling more complex queries. They offer many advantages for problems that our encoding scheme and algorithms are not able to solve.

However, one needs to reflect upon the use such heavyweight reasoning engines. Are they always useful? How much of their capabilities do we really need for a particular application? Can their algorithms be optimized for the problem at hand? Can these reasoning engines be deployed on any device we like? These are important questions that we need to ask ourselves and we have illustrated in this paper that sometimes the answer is clear. With a concrete problem such as those in section 2, we have shown that a dedicated algorithm, in this case for subsumption testing, can outperform a DL reasoner hands down.

It is more appropriate to compare our encoding scheme with some of the subtyping algorithms. The results are comparable. Some hierarchical encoding algorithms provide a more compact representation at the expense of re-encoding many classes in the hierarchy when conflicting codes arise if new classes need to be incrementally encoded. Our algorithm does not suffer from this issue, and is still able of doing the subsumption test based solely on the encoding of the classes or concepts at hand. Another nice advantage of having no conflicting codes at all is that the encoding of the multiple inheritance hierarchy can be carried out once in advance and reused later on by anyone who wants to. This is a major benefit for very large hierarchies, such as some of the ontologies

discussed in the paper, where, for example, mobile devices do not have the resources to do the encoding themselves.

The algorithm discussed in this paper does not cover all the aspects that are found in many description logic languages. Based on their usefulness for our applications and research projects we will investigate how the current encoding mechanisms can be extended without causing a big performance penalty if the requirements for inferencing support change.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[2] H. Ait-Kaci, R. S. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *Programming Languages and Systems*, 11(1):115–146, 1989.

[3] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web Service Semantics - WSDL-S, Version 1.0. http://www.w3.org/Submission/WSDL-S/, November 2005.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] M. Burstein, J. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. Website, November 2004.

[6] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 271–287, New York, NY, USA, 1993. ACM Press.

[7] Y. Caseau, M. Habib, L. Nourine, and O. Raynaud. Encoding of multiple inheritance hierarchies and partial orders. *Computational Intelligence*, 15:50–62, 1999.

[8] H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, December 2003.

[9] I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *WI '03: Proceedings of the IEEE/WIC International Conference on Web Intelligence*, page 75, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Cycorp. OpenCyc 1.0: Formalized Common Knowledge. http://www.opencyc.org/, August 2006.

[11] J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, U. Keller, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, H. Lausen, E. Oren, A. Polleres, D. Roman, J. Scicluna, and M. Stollberg. Web Service Modeling Ontology (WSMO). http://www.w3.org/Submission/WSMO/, June 2005.

[12] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on The What, Who, Where, When, and How of Context-Awareness, Conference on Human Factors in Computer Systems (CHI2000)*, 2001.

[13] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Deduction in concept languages: From subsumption to instance checking. *Journal of Logic and Computation*, 4(4):423–452, 1994.

[14] N. Drummond, M. Horridge, R. Stevens, C. Wroe, and S. Sampaio. Pizza Ontology v1.4 (2006/07/18). http://www.co-ode.org/ontologies/pizza/2006/07/18/, July 2006.

[15] P. Dusart. The kth prime is greater than k(ln k+ln ln k-1) for k≥2. *Mathematics of Computation*, 68:411–415, 1999.

[16] Electronic Commerce Code Management Association. Universal Standard Products and Services Classification V 11.0. http://eccma.org/unspsc/browse/, July 2002.

[17] D. Fensel, H. Lausen, J. de Bruijn, M. Stollberg, D. Roman, and A. Polleres. *Enabling Semantic Web Services : The Web Service Modeling Ontology*. Springer, 2006.

[18] Gene Ontology Consortium. the Gene Ontology project. http://www.geneontology.org/index.shtml, August 2006.

[19] T. R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[20] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang. An ontology-based context model in intelligent environments. In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference, San Diego, California, USA, January 2004.

[21] V. Haarslev and R. Möller. Racer: An OWL Reasoning Agent for the Semantic Web. In *Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems, in conjunction with the 2003 IEEE/WIC International Conference on Web Intelligence,Halifax, Canada, October 13*, pages 91–95, 2003.

[22] V. Kolovski, B. Parsia, Y. Katz, and J. A. Hendler. Representing web service policies in owl-dl. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 461–475. Springer, 2005.

[23] A. Krall, J. Vitek, and N. Horspool. Near optimal hierarchical encoding of types. In M. Aksit and S. Matsuoka, editors, *11th European Conference on Object Oriented Programming (ECOOP'97)*, pages 128–145, Finland, 1997. Springer.

[24] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press.

[25] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic Markup for Web Services, Release 1.1. http://www.daml.org/services/owl-s/1.1/overview/, November 2004.

[26] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview, February 2004.

[27] I. Niles and A. Pease. Towards a standard upper ontology. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 2–9, New York, NY, USA, 2001. ACM Press.

[28] N. F. Noy and D. L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.

[29] B. Parsia, V. Kolovski, and J. A. Hendler. Expressing ws-policies in owl. In *Policy Management for the Web Workshop, 14th International World Wide Web Conference*, 2005.

[30] B. Parsia and E. Sirin. Pellet: An OWL DL Reasoner. In *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, November 2004.

[31] Preuveneers, D., et al. Towards an extensible context ontology for Ambient Intelligence. In *Proceedings of the Second European Symposium on Ambient Intelligence*, volume 3295 of *LNCS*, pages 148–159. Springer, November 2004.

[32] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[33] L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Determining type, part, color, and time relationships. *IEEE Computer*, 16(10):53–60, October 1983.

[34] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding Semantics to Web Services Standards. In *Proceedings of the 1st International Conference on Web Services (ICWS'03)*, pages 395–401, June 2003.

[35] M. K. Smith, C. Welty, and D. L. McGuinness. OWL Web Ontology Language Guide, February 2004.

[36] R. Stevens, C. Goble, and S. Bechhofer. Ontology-based knowledge representation for bioinformatics, 2000.

[37] Strang. T., et al. CoOL: A Context Ontology Language to enable Contextual Interoperability. In *LNCS 2893: Proceedings of 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS2003)*, volume 2893 of *LNCS*, pages 236–247. Springer, November 2003.

[38] D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, Seattle, USA, August 2006.

[39] M. F. van Bommel and T. J. Beck. Incremental encoding of multiple inheritance hierarchies. In *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*, pages 507–513, New York, NY, USA, 1999. ACM Press.

[40] M. F. van Bommel and P. Wang. Encoding multiple inheritance hierarchies for lattice operations. *Data Knowl. Eng.*, 50(2):175–194, 2004.

[41] J. Vitek, N. Horspool, and A. Krall. Efficient type inclusion tests. In T. Bloom, editor, *Conference on Object Oriented Programming Systems, Languages & Applications (OOPSLA'97)*, pages 142–157, Atlanta, 1997.

[42] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung. Ontology Based Context Modeling and Reasoning using OWL. In *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, page 18, Washington, DC, USA, 2004. IEEE Computer Society.

[43] W. A. Woods. Understanding subsumption and taxonomy: A framework for progress. In J. F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, pages 45–94. Kaufmann, San Mateo, 1991.

[44] Y. Zibin and J. Gil. Efficient subtyping tests with PQ-encoding. In *Conference on Object-Oriented*, pages 96–107, 2001.

[45] Y. Zibin and Y. Gil. Subtyping tests (or type inclusion) benchmarks. http://www.zibin.net/subtyping-benchmarks.html, November 2001.