

Encoding Semantic Awareness in Resource-Constrained Devices

Davy Preuveneers and Yolande Berbers, *Katholieke Universiteit Leuven*

With the Semantic Web relying on ontologies to establish online machine-interpretable information, the Internet is growing into a semantically aware computing paradigm that facilitates Web entities' discovery of the knowledge and resources they need. Ambient intelligence aims to enable smart interaction beyond the Internet by

embedding intelligence into our environment to unobtrusively support users' daily activities. To accomplish these goals, ontologies and semantic awareness are crucial for better understanding a user's context.

While interest in the Semantic Web has spurred the development of large-scale semantic grid architectures, expanding the Semantic Web to the other side of the computing spectrum is a complex undertaking. The techniques and tools that support the Semantic Web aren't designed to deal with the resource-constrained devices with which people frequently interact in an ambient-intelligence environment.

To counter this disadvantage, we developed a coding scheme for ontologies that embeds semantic awareness in devices with limited memory and processing capabilities, such as sensory nodes and smart phones. This scheme provides a compact representation of an ontology and is enhanced with an efficient and effective semantic-matching algorithm.

Semantic matching and ambient-intelligence devices

Ambient intelligence envisions computing devices pervasively and unobtrusively embedded in

the environment, connecting hardware and software infrastructures, collecting information on users and their context, and adapting these devices' behavior to better assist the users. Ontologies help structure this information by using an established vocabulary of terms and concepts in a specific domain of interest. You can use them to describe the semantics of services offered in the environment, a device's capabilities, and a user's preferences and activities. In general, they characterize the situation of entities (person, place, or object) considered relevant to the interaction between a user and an application.¹

The OWL specification provides an expressive language with many logical constructs to define classes and properties and their relationships. Ontologies can reuse classes and properties defined in another ontology without having to remodel all the relationships that hold among them. The reuse of classes and properties by linking to other machine-interpretable ontologies facilitates the semantic matching and uniform interpretation of any shared information. Semantic matching is often accomplished through *subsumption* queries. Subsumption refers to the reflexive, transitive, and antisymmetric relationship between classes that states that a class *A* subsumes a class *B* if and only if the set of instances of class *A* includes the set of instances of

This coding scheme provides a compact representation of an ontology that ambient-intelligence devices can use effectively.

class *B*. The same principle holds for OWL properties.

The following scenario illustrates why location- and context-aware computing requires semantic matching:

An adventurous tourist is planning a backpacking trip and looking for new places to explore. His PDA with integrated GPS has become indispensable for finding nearby activities that match his interests and is handy for finding cheap accommodations and transportation. A local authority uses an e-tourism OWL ontology² to make travel information available on a web portal. A user profile on the PDA specifies the tourist's personalia and interests; the PDA will use this data to obtain personalized travel information. To facilitate ambient intelligence, an OWL context ontology on the PDA provides a semantic markup of the user profile, the user's context, and the link to the e-tourism ontology.³ To limit the search results when browsing the web portal on the PDA, only travel information is returned that semantically matches the tourist's context and preferences.

In this scenario, ontologies specify times and places of interest, classes of activities, transportation and accommodations, and relationships between them.

However, our experiments have shown that general-purpose ontology reasoners, such as Pellet,⁴ are unsuitable for manipulating an OWL context ontology on a smart phone to match services with the request, the user profile, and the preferences.³ Commodity computing hardware has several gigahertz of processing power for complex semantic-matching tasks. The smart phones and wireless programmable sensor nodes that are often in ambient-intelligence prototype environments must make do with much less.

To illustrate how limited these devices' processing power is, we benchmarked their performance with several well-known numeric tests in Java and native code, and compared the results with those of a fairly recent desktop. We used these platforms:

- a Qtek 9090 smart phone with a PXA263 400-MHz XScale CPU and 128 Mbytes of RAM for storing and running programs;
- an Imote 2 wireless sensor node with a PXA271 XScale CPU that dynamically scales from 13 to 416 MHz, 32 Mbytes of flash memory, and 32 Mbytes of SDRAM (synchronous dynamic RAM); and
- a desktop PC with a 3.2-GHz Pentium 4 processor and 1 Gbyte of memory.

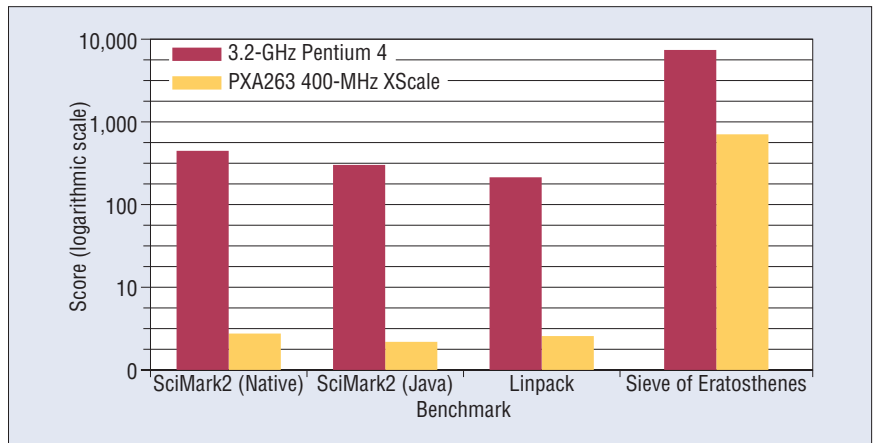


Figure 1. Comparing the performance of a smart phone (PXA263 400-MHz XScale CPU) with that of a desktop PC (3.2-GHz Pentium 4).

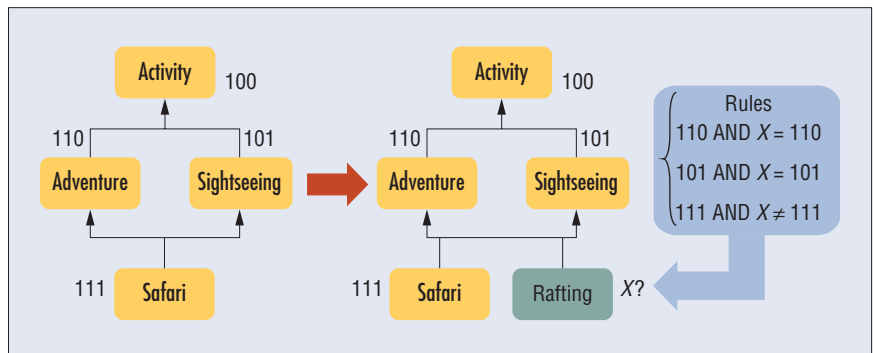


Figure 2. A hierarchy encoding of tourist activities with bit vectors.

As figure 1 illustrates, the speed of the CPU-bound numeric benchmarks on the smart phone was less than 1/100 of the speed on the desktop system. (The figure doesn't include results for the Imote2 because it performed similarly to the smart phone.) According to Moore's law, this performance lags by about four to eight years.

Subtype testing and inheritance encoding

The subsumption of classes in an ontology is somewhat related to multiple inheritance in an object-oriented programming language. Several inheritance-encoding algorithms for fast subtype testing⁵⁻⁸ inspired our encoding scheme because they offer two benefits:

- *Compact bit representation.* The data structures for representing subtype relationships are small. The algorithms usually encode each class as a bit vector and reuse bit positions to represent different classes. For example, the multiple-inheritance hierarchy in figure 2 on the

left shows four classes with a 3-bit encoding (we explain this figure in more detail later).

- *Fast matching.* Bit-vector representations allow for efficient, constant-time subsumption or subtype testing by means of binary AND/OR operations. For example, in figure 2, the class *Activity* subsumes the class *Safari* because $100 \text{ AND } 111 = 100$ (or $100 \text{ OR } 111 = 111$).

The encoding in figure 2 traverses the graph in breadth-first order (*Activity*, *Adventure*, *Sightseeing*, *Safari*) and assigns different bit positions from left to right to each root class (for example, *Activity*: 1..). It assigns a subclass a different bit position (for example, *Adventure*: .1. and *Sightseeing*: ..1) if that subclass has only one parent or has siblings (to distinguish subsumption by either the class itself or its parent or siblings). To compute the empty positions in the code, we use the binary OR operation on the codes of the subclass's parents. As the encoding continues, it pads the parents' bit vectors with zeros on the right (for example, *Activity*: 100).

Although the ontology-encoding requirements for efficient semantic matching are the same, we can't apply many of the subtype algorithms because programming languages and ontologies have a different point of view on classes. A compiler knows all the classes in a program (the closed-world assumption) and can assume that no other classes are being subsumed if they don't exist. Ontologies make no assumption on the information's completeness (the open-world assumption).

For example, in figure 2 on the right, a new ontology reuses the **Adventure** and **Sightseeing** classes to define the new **Rafting** subclass. However, we can't encode **Rafting** without modifying the existing codes. By inheriting from both the parent classes **Adventure** and **Sightseeing**, **Rafting** appears to also inherit from **Safari**. Because this isn't the case, the new subclass's encoding results in a conflict, and we must reencode **Safari**. To solve this conflict, we could introduce a new bit position for each class (that is, **Safari**: ...1. and **Rafting**: ...1), but this approach doesn't scale well for large ontologies. This solution would affect both the compact bit representation and the fast matching because it must compare more bits.

Reuse of ontologies and the concepts defined therein is a main goal of ontologies, so reencoding would occur rather often whenever conflicts arise owing to new inheritance relationships. Preferably, an ontology's encoding should be carried out once in advance and then shared as much as possible. So, we devised an algorithm that enables incremental, conflict-free encoding of multiple inheritance hierarchies of OWL classes and properties. It guarantees the reuse of existing encodings and the ability to extend encoded classes without leading to false subsumption results. The encoding is based on the multiplication of coprime numbers and on quickly detecting whether one number divides another one. To explain our coding scheme, we use subsumption relationships between classes. However, the same principles also hold for the subsumption of properties (for example, the property **hasSibling** subsumes the property **hasSister**) and anonymous classes (for example, intersections of class descriptions, or domain and range constraints for a property used in a class extension).

Ontology encoding with prime numbers

Incremental encoding without conflicts requires that each class in the hierarchy has its

own identifier. We call this identifier the *differentiating gene* g_A of a class A in the inheritance (or subsumption) hierarchy χ . Assume that a function $\varphi : \chi \rightarrow G$ uniquely maps a set of classes in a hierarchy $\chi = \{C_1, C_2, \dots, C_n\}$ to a set of genes $G = \{g_1, g_2, \dots, g_n\}$:

$$\forall C \in \chi : g = \varphi(C)$$

Inspired by nature but with a twist, we define that a class A that's subsumed by a class B (denoted as $A < B$) inherits all the genes of class B . As such, a class A can have multiple genes: those that it inherits from its ancestors, and g_A , which it shares with its descendants. We define $\Gamma(A)$ as the set of genes of class $A \in \chi$ with ancestor and equivalent classes $B \in \chi$ (we denote ances-

Reuse of ontologies and the concepts defined therein is a main goal of ontologies, so reencoding would occur often when conflicts arise owing to new inheritance relationships.

try as *Child <: Ancestor* and equivalence as *Father \equiv MaleParent*):

$$\Gamma(A) = \{g_A\} \cup \{\Gamma(B) \mid A <: B \vee A \equiv B\} \quad (1)$$

If classes $A, B \in \chi$ are equivalent, then this definition ensures that $\Gamma(A) = \Gamma(B)$. We need to determine a function $\psi(\Gamma)$ that encodes a set of genes $\Gamma(A)$ to a compact representation $\gamma(A)$,

$$\forall A \in \chi : \gamma(A) = \psi(\Gamma(A))$$

such that either the maximum or the average encoding length $|\gamma(A)|$ is minimal; that is,

$$\max_{A \in \chi} |\gamma(A)| \quad \text{or} \quad \sum_{A \in \chi} |\gamma(A)| \quad \text{is minimal}$$

The gene thus represents an inheritance or subsumption relationship. The goal now is to find an efficient coding scheme that provides a compact representation of these relationships. A straightforward but naive and memory-consuming implementation for n classes uses a distinguishing bit position for each gene (or class) and represents in-

heritance in a binary matrix of n vectors, each n bits long, which thus requires n^2 bits in total. Our coding scheme achieves much better results.

Compact representation

The number of classes in an ontology is usually much larger than the number of a class's ancestors. Consequently, the binary-matrix representation of the inheritance relationships is often sparse; that is, it contains many 0s. So, we'll only encode a reference to the ancestors in the representation of a class. To do this, we assign each class a number as a differentiating gene $g \in G$ that's coprime with all the other genes—that is,

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

We can then define the encoding $\gamma(A)$ of class A as the multiplication of its differentiating gene g_A with the inherited genes of its ancestors—that is, the product of the genes in $\Gamma(A)$:

$$\gamma(A) = \prod_{g \in \Gamma(A)} g \quad (2)$$

The following theorem illustrates the importance of using coprime numbers as genes.

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 ,*

$$A < B \iff \gamma(A) \bmod \gamma(B) = 0$$

PROOF: *We provide a proof for the implication in both directions:*

$$[\implies] : A < B \implies \gamma(A) \bmod \gamma(B) = 0$$

If a class $A \in \chi$ is subsumed by a class $B \in \chi$, then by definition

$$A < B \iff A \equiv B \vee A <: B$$

If $A \equiv B$, then by using the definition in equation 1, we conclude that both classes have the same encoding:

$$\gamma(A) = \gamma(B) \implies \gamma(A) \bmod \gamma(B) = 0$$

If $A <: B$, then class A inherits all genes from class B , or $\Gamma(B) \subset \Gamma(A)$. Using the definition in equation 1,

$$\Gamma(B) \subset \Gamma(A) \implies \gamma(A) = q \cdot \gamma(B)$$

with $q \in \mathbb{N}$ defined as

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

This means that $\gamma(A) \bmod \gamma(B) = 0$:

$$[\Leftarrow] : A < B \Leftarrow \gamma(A) \bmod \gamma(B) = 0$$

If the encoding of class B divides that of class A , then

$$\gamma(A) = q \cdot \gamma(B) \text{ with } q \in \mathbb{N}$$

According to equation 2, we can define the encodings of classes A and B as

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

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

Combining these definitions results in

$$\gamma(A) = q \cdot \gamma(B) \Rightarrow \prod_k g_k = q \cdot \prod_l g_l$$

Each g_l on the equation's right divides the product on the equation's left. Because all $g \in G$ are pairwise coprime, there's a k for each l such that $g_k = g_l$, or

$$\Gamma(B) \subset \Gamma(A) \Rightarrow A < B$$

As a corollary, 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 ,

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

The genes $g \in G$ needn't be prime numbers. For example, the genes in $G = \{76, 93, 145, 161, 169, 187\}$ are pairwise coprime but aren't prime numbers. However, because each prime $p \in P$ can be present in the factorization of only one gene, using primes as genes results in shorter encodings. A class's encoding length—computed using the binary logarithm $\lg(x)$ —is always smaller than that of its descendants: A class B with an ancestor class A has at least one extra gene g_B in its factorization. Because the smallest prime number is 2, the multiplication of the genes of class B will be at least 1 bit longer than the encoding of class A .

Figure 3 illustrates the encoding of a multiple-inheritance hierarchy $\chi = \{A, R, H, B, L\}$. Each class's differentiating gene is underlined. For example, class B inherits genes 2 and 7 from its ancestors and is as-

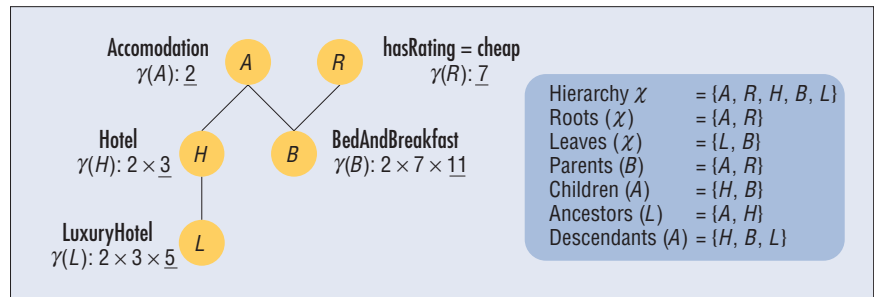


Figure 3. A hierarchy encoding of accommodation with prime numbers.

signed 11 as its differentiating gene g_B . The encoding of class B becomes $\gamma(B) = 2 \times 7 \times 11 = 154$. Because classes A and R have no ancestors, their encodings $\gamma(A)$ and $\gamma(R)$ are based solely on their differentiating genes $g_A = 2$ and $g_R = 7$. The degree of compaction depends solely on the prime number p assigned to a class C . Later, we discuss the heuristics for assigning a prime number to each class to optimize the encoding length.

Conflict-free and fast incremental encoding

Because we assign each class $C_i \in \chi$ a distinct prime number $p_i = \varphi(C_i)$ as its differentiating gene, subsumption-testing conflicts never arise. To encode a new class C_{n+1} that's incrementally added to a hierarchy χ of n classes $C_{1..n}$, we assign the next unused prime $p_{n+1} = \varphi(C_{n+1})$. Because conflicts are of no concern, encoding leaf classes is straightforward. Moreover, there's no need to traverse the hierarchy to collect the ancestors' genes. Instead, we can compute the new code for class C_{n+1} using the least common multiple of its parents' encoding. If C_{n+1} is added to the hierarchy χ as a nonleaf class, we update its descendants' encoding by multiplying that encoding by p_{n+1} and the number of genes of any new ancestors. This reflects the descendants' inheritance from C_{n+1} and its ancestors.

Optimizing the encoding length

We present two heuristics for the mapping function $p = \varphi(C)$, which assigns a prime number p to each class C to achieve compaction comparable to that of the subtyping algorithms we mentioned before. Assigning a random prime number to classes isn't a good idea because a class's prime number affects the encoding length of not only the class itself but also its descendants. As a rule of thumb, we assign a prime number to a class before assigning one to its descendants. By intelligently ordering the re-

maining candidate classes, we can achieve a more compact representation.

Heuristic 1 attempts to minimize the average encoding length of the bit vectors in the subsumption hierarchy. It sorts the candidate classes by their number of descendants and assigns a prime number to the class with the most descendants. This heuristic results in the overall smallest encoding of the whole subsumption hierarchy. Figure 4 shows a pseudocode implementation of this hierarchical-encoding algorithm. The hierarchy containing the classes C serves as input; the output is the encoding $\gamma(C)$ for each class C .

Heuristic 2 attempts to minimize the bit vectors' maximum encoding length. It tries to keep the longest code's size as short as possible. This longest code will occur in one of the hierarchy's leaf classes. This heuristic replaces the `bestClass` selection in the `EncodeHierarchy()` algorithm (see figure 4) with three steps:

1. Given the partial gene assignment, determine each leaf class's minimum code. If k of a leaf class's ancestors haven't yet obtained a prime number, then compute this minimum code by multiplying the already inherited genes with the next available $k + 1$ prime numbers.
2. Of all the candidate classes that have the leaf class with the largest minimum code as a descendant, select the class with the most descendants as the best class.
3. Remove this class from the list, and add to the list each of its children whose ancestors have already been assigned a differentiating gene.

This heuristic achieves the smallest maximum encoding length for a single class.

For both heuristics, figure 5 presents the encoding-length results for several ontologies and inheritance hierarchies:

```

EncodeHierarchy(in: hierarchy, out: gamma)
-----
i = 0, n = SizeOf(hierarchy)
primeTable[1..n] = ComputePrimes(n)
classList = Roots(hierarchy)
while SizeOf(classList) > 0 do {
  // Determine the next best class
  bestClass = First(classList)
  for each C in classList do
    if SizeOf(Descendants(C)) > SizeOf(Descendants(bestClass)) then
      bestClass = C

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

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

Figure 4. A pseudocode implementation of the minimal-average-encoding-length algorithm.

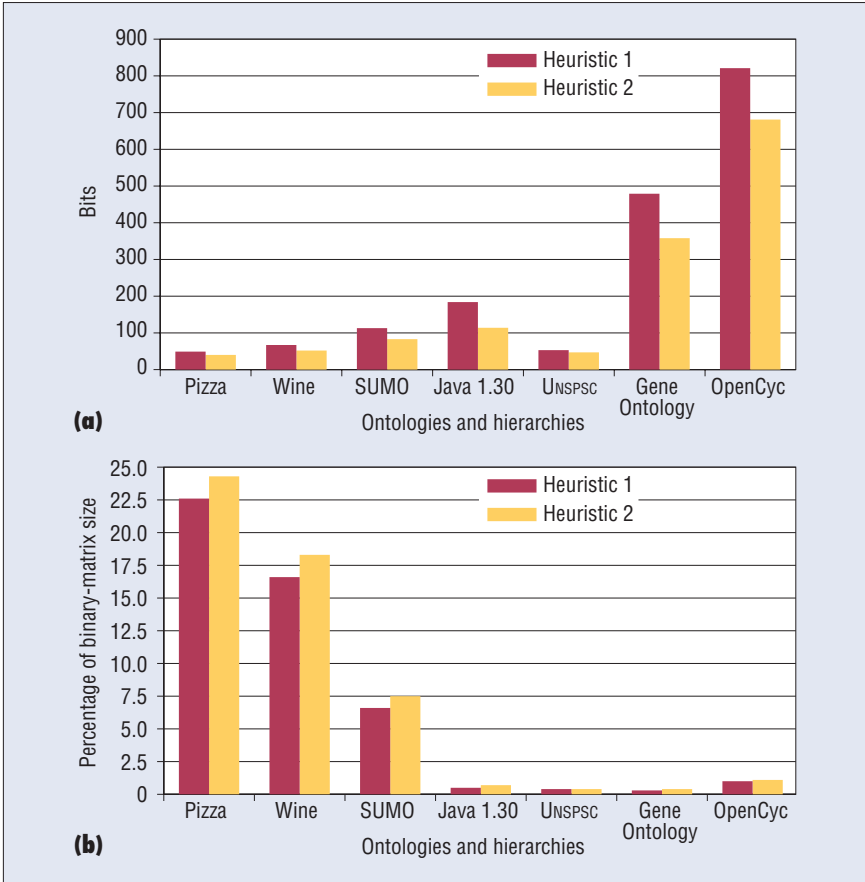


Figure 5. The (a) maximum class encoding lengths and the (b) total encoding lengths relative to the binary matrix's size, for several ontologies and hierarchies.

- The *Pizza* ontology is used in the Protege-OWL tutorial; it was contributed by the CO-ODE (Collaborative Open Ontology Development Environment) project (99 classes).
- The well-known *Wine* ontology is used to explain the basic features and some of the more advanced concepts of OWL DL (133 classes).
- *SUMO* (the Suggested Upper Merged Ontology) models general-purpose knowledge; it's developed by the IEEE Standard Upper Ontology Working Group (630 classes).
- The *Java 1.30 Types* hierarchy isn't an ontology but rather part of a benchmark suite for comparing the compactness of several subtyping algorithms (5,438 classes).
- *UNSPSC* (the Universal Standard Products and Services Classification) is a taxonomy of products and services (9,797 classes).
- The *Gene Ontology* project is a joint effort to create a vocabulary of genes of all organisms and to describe associated gene products (20,945 classes).
- *OpenCyc* is an upper ontology that aims for human-like reasoning with a general knowledge base of everyday common sense. We used the v0.7.8b OWL version (25,565 classes).

Figure 5a shows the length of the longest encoding among all classes. Figure 5b shows the total length of all encodings added together relative to the binary matrix's size. Remember that for a binary-matrix representation of n classes, a single bit vector's length is n bits and the total matrix is n^2 bits.

Optimizing the subsumption test

In theorem 1, we argued that a class $A \in \chi$ is subsumed by a class $B \in \chi$ if and only if the gene $g_B = \phi(B)$ of class B divides the encoding $\gamma(A)$ of class A . Given the sizes of the bit vectors in figure 5a, a machine word length of 32 or 64 bits clearly won't suffice. We need arithmetic support for big integers. However, after investigating several math libraries that support arbitrary precision, we concluded that the division algorithms have some avoidable overhead:

- Most libraries only support divisions with two big integers. However, more than

100,000,000 prime numbers fit in 32 bits, and an ontology with that many classes won't occur anytime soon. So, a faster implementation with the divisor being a machine-word integer is possible.

- The division operation takes into account the sign of the dividend and the divisor. Because both integers are positive, we don't need to compute the sign of the result.
- Many libraries compute the quotient and the remainder at the same time. Because we need to know only whether the division succeeds, we don't need to compute the quotient.

So, we implemented a fast method for checking whether a big integer can be divided by an integer of fewer than 32 bits. In many cases, our coding scheme lets us decide whether a class A (with encoding $\gamma(A)$ and differentiating gene p_A) subsumes a class B (with encoding $\gamma(B)$ and differentiating gene p_B) without carrying out the expensive division. We use the following three tests to quickly rule out subsumption.

Comparing bit-vector lengths

A class A will never subsume a class B if its encoding length is larger than that of class B . This is because a descendant of class A always inherits all the genes from class A and has its own differentiating gene.

Comparing differentiating prime numbers

Because our scheme encodes classes before their descendants, we can rule out subsumption of class B by class A if $p_A > p_B$. We could also use the depths of classes A and B in the hierarchy by ruling out subsumption of class B by class A if $depth(B) < depth(A)$, but experiments have shown that this doesn't much decrease the number of divisions.

Comparing division by the principal prime numbers

Not all divisions can be avoided with the optimizations we've described. Because a big integer might be several hundred bits long, the implicit computation of the remainder might be a lot slower than the simple binary OR or AND operations that subtype-testing algorithms use. Nevertheless, we can use a similar technique for fast subsumption checking between classes. For any prime number $p \in P$, a class A won't sub-

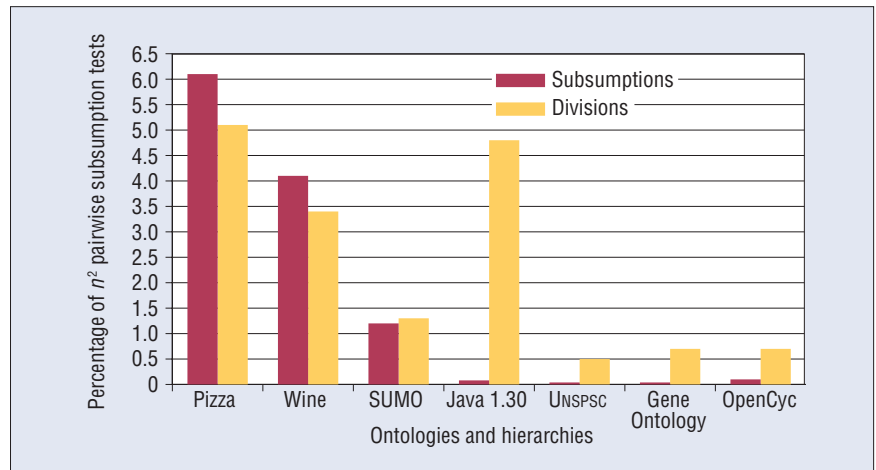


Figure 6. Pairwise subsumption of all n classes in the hierarchy.

sume a class B if a prime number divides the encoding of class A but not that of class B . So, a class A doesn't subsume a class B if

$$\gamma(A) \bmod p = 0 \wedge \gamma(B) \bmod p \neq 0$$

If each class describes in k bits whether k strategically chosen prime numbers occur in its factorization, then k of these comparison tests for two classes A and B can execute in parallel with a simple binary AND operation on both k -size bit vectors. Of course, to maximize the benefit, these prime numbers p must be well chosen. For example, a prime number of a single root class that's inherited by all the other classes will gain nothing. We call these k well-chosen prime numbers the k principal prime numbers of a specific hierarchy encoding. We've discussed elsewhere how to determine these numbers.⁹

Evaluating the tests

Figure 6 shows the results of an experiment in which we conducted n^2 subsumption checks—that is, each combination of the n classes in the subsumption hierarchy. The results show for $k = 64$ how many subsumption tests returned true (that is, there was a semantic match) and how many divisions were needed. For some hierarchies, the quicker comparison tests reduced the number of divisions to less than 1 percent of the subsumption tests. When two classes semantically match, the quicker comparison tests can't rule out subsumption. In that case, divisions will be necessary to verify the semantic match. So, the number of divisions should always be higher than the number of subsumptions. The only case where a division is not needed to verify the match is for semantically equivalent classes

in an ontology. Because they get the same distinguishing gene, we once again avoid a division.

Evaluating the heuristics

Because ontology encoding is time- and memory-intensive, we carried out this step on a desktop PC. The advantage is that it needs to be done only once. After the encoding, we tested our semantic-matching algorithm for the tourism scenario we mentioned earlier, using the Qtek 9090 smart phone and the Imote2 sensor node. We also compared our results with several subtype-encoding algorithms.

Setting up the test environment

We used Pellet on the desktop PC to classify the concepts and to infer the implicit subsumption relationships among the classes and properties in three ontologies:

- *SUMO*.
- *UNSPSC*. Part of the UNSPSC ontology deals with tourism and travel-related services, but we encoded the entire ontology for the experiment.
- *Tourism*. This ontology imports various concepts from tourism ontologies (<http://e-tourism.derit.at/ont/e-tourism.owl> and <http://protege.cim3.net/file/pub/ontologies/travel/travel.owl>) as well as our own context ontology (www.cs.kuleuven.be/~davy/ontologies/2007/04/Context.owl).

We then encoded the inferred subsumption hierarchy into a compact representation. We stored the encoded ontologies on the resource-constrained devices along with Java and native C implementations of

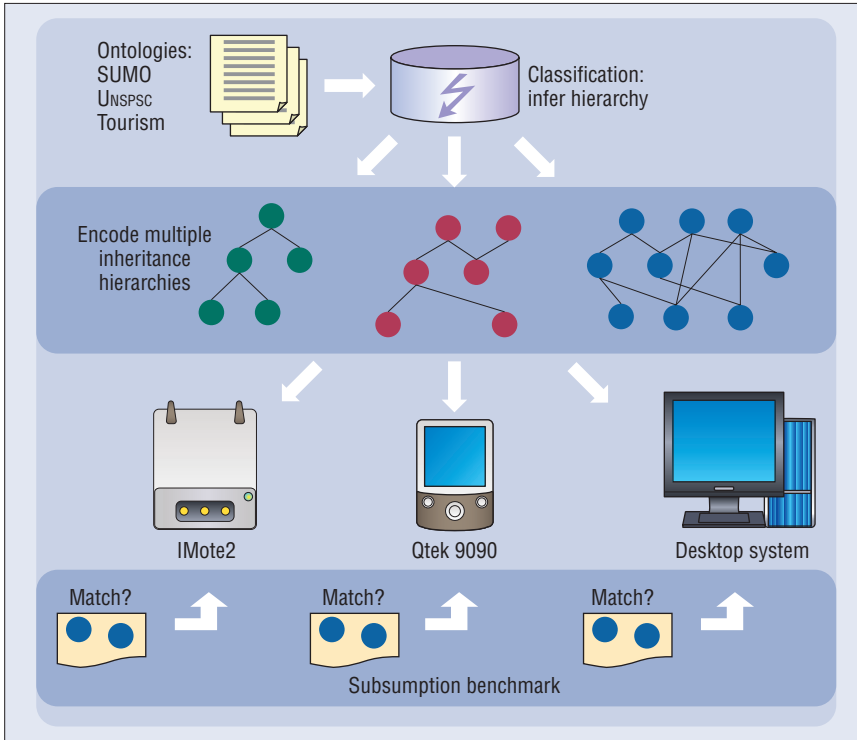


Figure 7. The semantic-matching test environment.

The Authors



Davy Preuveneers is a PhD student and research assistant in the DистриNet research group of the Katholieke Universiteit Leuven's Department of Computer Science. His research is in middleware and service-oriented architectures for context-aware service interaction and adaptation, particularly in mobile- and ubiquitous-computing environments. He received his MSc in computer science and MSc in artificial intelligence from the Katholieke Universiteit Leuven. Contact him at the Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Leuven), Belgium; davy.preuveneers@cs.kuleuven.be; www.cs.kuleuven.be/~davy.



Yolande Berbers is an associate professor in the Katholieke Universiteit Leuven's Department of Computer Science and a member of the DистриNet research group. Her research interests include software engineering for embedded software, ubiquitous computing, service architectures, middleware, real-time systems, component-oriented software development, distributed systems, environments for distributed and parallel applications, and mobile agents. She received her PhD in computer science from the Katholieke Universiteit Leuven. She's a member of the IEEE. Contact her at the Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Leuven), Belgium; yolande.berbers@cs.kuleuven.be; www.cs.kuleuven.be/~yolande.

our semantic-matching algorithm. Then we tested how many subsumption checks were carried out per time unit. Figure 7 illustrates the setup's workflow.

Benchmarking algorithm performance

Figure 8a compares the encoding lengths

for our scheme with those for the hierarchy-encoding algorithms of Andreas Krall and his colleagues⁷ and Yves Caseau and his colleagues.⁶ These two other algorithms gave the shortest encodings of all the algorithms we tested. Our scheme achieves a less compact representation, but it doesn't need to reencode classes in the hierarchy

when new classes are added. We carried out performance testing on both the Qtek 9090 and the Imote2 but show the results only for the Qtek. The Imote2 has a similar processor with comparable speed and gives similar results. The slight differences that were observed could be due to the use of a different cross-compiler for the Linux-based Imote2.

We also loaded the original ontologies into Pellet on the desktop PC and measured the memory consumption. Given that Pellet's memory usage was on the order of tens of megabytes instead of kilobytes with our encoding, the memory usage of an ontology reasoner clearly would be a bottleneck for resource-constrained devices.

Figure 8b shows the number of subsumption tests per millisecond. The first experiment compared our algorithm's Java implementation with a Java HashSet alternative that stores each concept's ancestors in a hash table. For all the ontologies, our algorithm was quicker. Moreover, because each concept in the HashSet implementation had a pointer to all its ancestors (requiring at least 32 bits per pointer) and some overhead for structuring the hash table, our algorithm consumed less memory.

The second experiment compared our algorithm's C implementation with a native implementation of Krall and his colleagues' algorithm. Although our algorithm is slower, it doesn't perform that much worse. We also tested our algorithm on the desktop PC and compared its performance with that of the ontology reasoner. The Java version achieved more than 20,000 requests per millisecond; the C version was on average approximately five times quicker. Pellet could process only approximately 80 subsumption tests per millisecond. Our algorithm was about 250 times faster than this Java-based reasoner. Other experiments with native ontology reasoners, such as FaCT++ and Racer, gave results in the same order of magnitude (fewer than 100 subsumption tests per millisecond).

These extraordinary results were achievable only because we compiled the knowledge ahead of time when we constructed the subsumption hierarchy. Nonetheless, if your primary concern is the semantic matching of service descriptions, user preferences, and device capabilities (or of any user context, for that matter), and the corresponding ontologies are known in advance, then you can significantly improve the semantic matching by transforming the ontology

with our scheme. The coding itself can be time consuming for very large ontologies. However, this is acceptable because, as we mentioned before, encoding should only be carried out once. The results show that embedding semantic awareness in mobile and sensory devices with limited memory and processing capabilities is feasible, even when applications run in a Java virtual-machine environment.

Our semantic-matching algorithm offers an interesting alternative to class-subtyping encoding techniques, because it yields a new way of compaction without changing old conflicting codes during incremental encoding.

If you're interested only in subsumption checking, there are better approaches than ontology reasoners, which consume considerable memory and processing time. But comparing an ontology reasoner with a vastly optimized and dedicated algorithm for a single problem such as subsumption would be unfair. Ontology reasoners prove their usefulness for complex queries that our algorithm currently can't solve. For example, our algorithm can't test a class's satisfiability—that is, determine whether instances of a particular class can actually ever exist.

While our algorithm can encode classes and properties, including restrictions on a property's domain and range, it doesn't cover all the logical constructs that are in many description-logic languages such as OWL. On the basis of the changing requirements for inferencing support, we'll investigate how to extend our encoding mechanisms for other logical constructs without seriously decreasing our algorithm's performance. ■

References

1. A.K. Dey and G.D. Abowd, "Towards a Better Understanding of Context and Context-Awareness," presented at the CHI 2000 Workshop the What, Who, Where, When, and How of Context-Awareness, 2000, <http://smartech.gatech.edu/bitstream/1853/3464/5/00-18e.pdf>.
2. J. Cardoso, "Developing an Owl Ontology for E-tourism," *Semantic Web Services, Processes and Applications*, J. Cardoso and A.P. Sheth, eds., Springer, 2006, pp. 247–282.

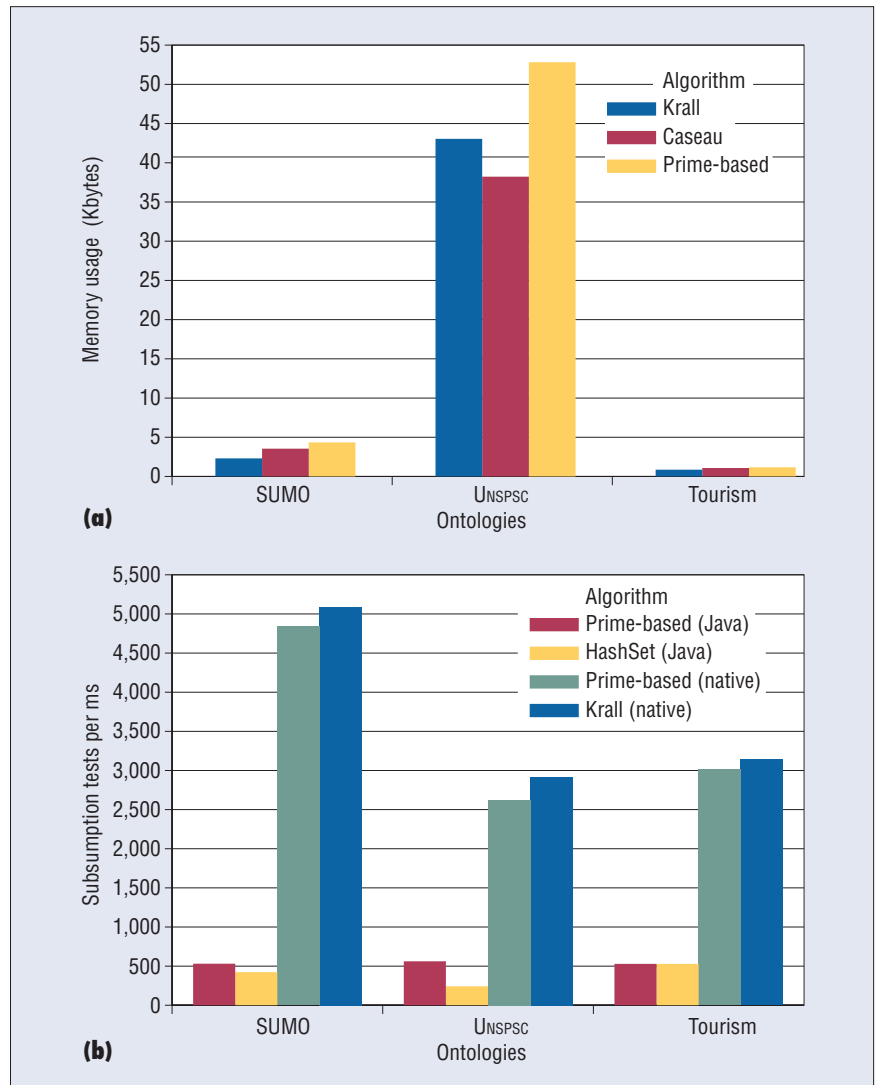


Figure 8. Subsumption testing with the prime-based algorithm: (a) encoding length, (b) performance.

3. D. Preuveneers et al., "Towards an Extensible Context Ontology for Ambient Intelligence," *Proc. 2nd European Symp. Ambient Intelligence*, LNCS 3295, Springer, 2004, pp. 148–159.
4. E. Sirin et al., "Pellet: A Practical OWL-DL Reasoner," *J. Web Semantics*, vol. 5, no. 2, 2007, pp. 51–53.
5. R. Agrawal, A. Borgida, and H.V. Jagadish, "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases," *Proc. 1989 ACM SIGMOD Int'l Conf. Management of Data (SIGMOD 89)*, ACM Press, 1989, pp. 253–262.
6. Y. Caseau et al., "Encoding of Multiple Inheritance Hierarchies and Partial Orders," *Computational Intelligence*, vol. 15, no. 1, 1999, pp. 50–62.
7. A. Krall, J. Vitek, and N. Horspool, "Near Optimal Hierarchical Encoding of Types," *Proc. 11th European Conf. Object Oriented Programming (Ecoop 97)*, Springer, 1997, pp. 128–145.
8. M.F. van Bommel and T.J. Beck, "Incremental Encoding of Multiple Inheritance Hierarchies," *Proc. 8th Int'l Conf. Information and Knowledge Management (CIKM 99)*, ACM Press, 1999, pp. 507–513.
9. D. Preuveneers and Y. Berbers, *Prime Numbers Considered Useful: Ontology Encoding for Efficient Subsumption Testing*, tech. report CW464, Dept. of Computer Science, Katholieke Univ. Leuven, 2006.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.