Ontology Versions Management on the Semantic Web

Hong-Won Yun, *Member, KIMICS*

Abstract—In the last few years, The Semantic Web has increased the interest in ontologies. Ontology is an essential component of the semantic web. Ontologies continue to change and evolve. We consider the management of versions in ontology. We study a set of changes based on domain changes, changes in conceptualization, metadata changes, and temporal dimension. In many cases, we want to be able to search in historical versions, query changes in versions, retrieve versions on the temporal dimension. In order to support an ontology query language that supports temporal operations, we consider temporal dimension includes transaction time and valid time. Ontology versioning brings about massive amount of versions to be stored and maintained. We present the storage policies that are storing all the versions, all the sequence of changed element, all the change sets, the aggregation of change sets periodically, and the aggregation of change sets using a criterion. We conduct a set of experiments to compare the performance of each storage policies. We present the experimental results for evaluating the performance of different storage policies from scheme 1 to scheme 5.

Index Terms—Ontology, Version Management, Semantic Web, Storage Policy

I. INTRODUCTION

Semantic Web has been presented as the next step in the evolution of the World Wide Web. Research on ontology is becoming increasingly widespread in the computer science community. Ontologies have also become important in the Semantic Web. Purposes of ontology on the web are making the knowledge about a particular domain explicit, sharing and reusing this knowledge, and analyzing domain knowledge. Ontology development is still difficult and time-consuming.

Ontology is an explicit specification of a conceptualization of a domain. Ontologies are often seen as basic building blocks for the Semantic Web. Ontologies continue to change and evolve over time. A major change in ontologies are caused by domain changes and concept changes. Domain changes and concept changes in the shared conceptualization require modifications of the ontology. The causes of changes are classified as changes in the domain, changes in conceptualization, or changes in the explicit specification [1-7].

Ontology versioning is related to changes in ontologies. More properly, an ontology versioning consists in a collection of ontology versions. In general it can be said that there is a lack of methods managing ontology. To understand the problem of reusing and evolving ontologies in the Semantic Web, we consider the management of versions in ontology. In this paper, we study a set of changes based on domain changes, changes in conceptualization, metadata changes, and temporal dimension. We use the two most common aspects are valid time and transaction time in order to support a temporal ontology query.

Ontology versioning brings about massive amount of versions to be stored and maintained. To management massive data, efficient storage policies are necessary. Some of version management methods have been developed; these methods can be used to manage versions in different document models. These schemes are unlikely to be appropriate for ontology versions management. We propose several storage policies for ontology versions management and evaluate them.

The rest of the paper is organized as follows. Section 2 introduces updates and changes in ontologies. Section 3 describes the change specification considering time dimension. Section 4 presents several storage policies for ontology versions management. Our experimental results are described in Section 5. Conclusions can finally be found in Section 6.

II. ONTOLOGY CHANGE

One widely cited definition of an ontology is Gruber's [6]. According to Gruber [6], an ontologies is a specification of a shared conceptualization of a domain. Ontologies continue to change and evolve. In [4], Changes in ontologies are caused by either: changes in the domain, changes in the shared conceptualization, or changes in the specification. In [8, 9], It is stated that an ontology has classes, class hierarchy, instances of classes, slots as first-class objects, slot attachments to class to specify class properties, and facets to specify constraints on slot values. Those elements involve ontology change.

Wiederhold [10] describes four types of domain differences: (1) terminology: different names are used for the same concepts, (2) scope: similar categories may not match exactly; their extensions intersect, but each may have instances that cannot be classified under the other, (3) encoding: the valid values for a property can be different, even different scales could be used, (4) context: a term in one domain has a completely different meaning in another.

Manuscript received February 17, 2004.

This article was supported by the research grant 2002 of Silla University. Hong-Won Yun is Associate Professor in Silla University, Busan, Korea.(Tel: +82-51-999-5065; e-mail: hwyun@silla.ac.kr)

OntoView [11] is the ability to compare ontologies at a structure level. OntoView have the comparison function distinguish between the following types of change: non-logical change, logical definition change, identifier change, addition of definitions, deletion of definitions. OntoView deal with four types of change.

The research in ontologies started with defining what a formal ontology, shifted to the development of representation languages, and developed the method of evolution and versioning [7]. The complexity of ontology evolution increases as ontologies change and evolve, so a systematic ontology management is required.

We examine a number of ways to represent these change information for an ontology versions. A number of research works were done on ontology change, whereas through studies concerning change specification are still lacking. In this paper, we deal with the change specification in an ontology versioning, also taking into account temporal dimension aspects.

Large ontologies are essential components in web service systems. As ontologies become larger and longer lived, an amount of versions to be stored and maintained become massive volume. We will study the storage policies using the change specification. Change sets are used as logs in database to maintain versioned data, so we will show a proper policy taking into account both performance and storage space usage. Also, we present the experimental results for evaluating the performance of different storage policies.

III. CHANGE SPECIFICATION

In this section, we introduce a change specification for changes in ontologies. Our change specification is represented by a set of changes. We assume that a set of changes consists of instance data change, structural change, identifier change, meta data change, and temporal dimensions. Instance data change in ontology is comparable to update in database instances, e.g., change of a class properties, update of a slot values or restrictions, etc. Structural changes occur when classes are added, removed or moved. Identifier change means a rename of element in an ontology. Meta data describes comments about the change. Temporal dimension includes transaction time and valid time.

The *valid time* of a fact is the time when the fact is true in the modeled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. A database fact is stored in a database at some point in time, and after it is stored, it is current until logically deleted. The *transaction time* of a database fact is the time when the fact is current in the database and may be retrieved [12].

To formally represent a set of changes, we use the symbols as following:

- D : instance data change
- S : structural change
- I : identifier change
- M : meta data change
- T : temporal dimensions

We define that a set of changes contains four elements are related to change and time dimension T.

- $C = (D \cup S \cup I \cup M) \circ T$

Suppose C_n is a set of changes for n^{th} and V_n is a version for n^{th}. A set of changes C_{n+1} applied to V_n that result in V_{n+1}, i.e. $V_{n+1} = V_n \circ C_{n+1}$. Figure 1 shows a relation between versions and change sets. In figure 1, C_1 or C_2 means a set of changes. D and S mean that instance data change and structural change occurred respectively. For instance, we can apply C_1 to Version 0 and then we can produce Version 1. C_1 contains a list of specific operations related to instance data change. C_2 contains a list of structural change operations.

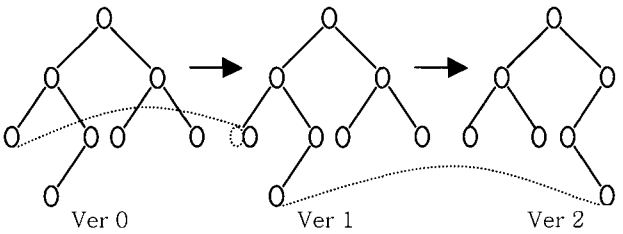


Fig. 1 Relation between version and change set.

Table 1 shows list of change sets. In table 1, an *Eid* identifies uniquely a particular element in an ontology. The changing object is described with *ObjectName*. The temporal dimension is the core of ontology versioning to support the temporal query. The temporal dimension includes valid time and transaction time as following:

- T = { *Eid, vt, tt* }

(1) vt = [*valid_start_time, valid_end_time*]

(2) tt = [*transaction_start_time, transaction_end_time*]

Table 1 List of change sets

Category	Set of change
Instance data change	D = { *Eid, ObjectName, OperationName, OldValue, NewValu e*}
Structural change	S = { *Eid, OperationName, OldName, NewName* }
Identifier change	I = { *Eid, OperationName, OldEid, NewEid* }
Meta data change	M = { *Eid, ObjectName, OperatioName, OldValue, NewValue* }
Temporal dimensions	T = { *Eid, vt, tt* }

The time dimension naturally has hierarchy as year, month, week, and day. Here we have years as the coarsest granularity and days as the finest granularity.

Table 2 gives operations and semantics the corresponding to *OperationName* in the change sets. In Table 2, e is an identifier *Eid*, *nv* is a new value after update and *ov* is an old value previous update.

Table 2 Basic operations and semantics in change sets

Operations	Semantics
UpdateValue (*e, nv, ov*)	Modifies *ov* the instance of the element *e* to *nv*
DeleteValue (*e*)	Deletes the instance of the element *e*
InsertValue (*e, nv*)	Inserts *nv* the instance of the element
RenameValue (*e, ne, oe*)	Renames *oe* the name of the element to *ne*
MoveValue (*e, n, p*)	Moves the value in the class *p* to be the value of class of *e* in position *n*
CreateClass (e, *n, p*)	Creates a superclass *e* in position *n*
MoveClass (*e, n, p*)	Moves the class rooted in the class *p* to be the class of *e* in position *n*
DeleteClass (*e*)	Deletes the class rooted in the class *e*
InsertClass (*e, n, C*)	Inserts the class *C* as a subclass of the class *e* in position *n*
UnionClass (*e*, e_1, e_2)	Merges the class e_1 and the class e_2 results in the class rooted in the class *e*
DivideClass (*e*, e_1, e_2, n_1, n_2)	Divides the class e into the class e_1 and the class e_2, and inserts each class in position n_1, n_2
RenameId (*e, ne, oe*)	Renames *oe* the identifier of class to *ne*

As we mentioned previous, a version is represented by change sets. In order to generate a version from another one, we maintain change sets. To generate a particular version, change sets are applied to previous stored version. A set of changes has temporal dimension to process temporal queries. Examples of temporal queries are: what are the product's names update since time *t*? When did the car Hyundai Elantra add the directory? The way for querying the history is to search change sets includes that information and reconstruct the version.

IV. STORAGE POLICIES

In previous section, we described the change specification for changes in ontologies. We discuss the storage policies of versioned data in this section. We need to store the first version in the ontology repository s well as the last version. The first version can be used to create particular version. Several storage policies may be considered as following:

- Scheme 1 – Storing all the versions
- Scheme 2 – Storing all the sequence of changed elements
- Scheme 3 – Storing all the change sets
- Scheme 4 – Storing the versions and the aggregation of change sets periodically
- Scheme 5 – Storing the versions and the aggregation of change sets using a criterion

Scheme 1 is that all the snapshots of the ontology are stored in the storage. This scheme is required lots of storage space. There is no additional processing overhead to generate queried versions. Similar this scheme is already proposed to manage versions in different document representation models. We use it to compare with other schemes. Scheme 2 is the policy that stores only changed elements, e.g., store a history of an ontology. This storage policy use linked list to maintain versioned data. To process temporal query in this scheme, it is needed to search a link. It is not required to generate any historical versions.

Whenever versions are created, scheme 3 stores previous the sequence of change but not versions. The sequence of change sets from the earliest versions to the current versions is stored. To access to historical information, this scheme need to generate the versions. We can aggregate the change sets periodically and store them. Scheme 4 stores the aggregation of change sets and the versions periodically. Aggregations of change sets are obtained via dividing all the sequence of change sets periodically.

Also, we can divide the sequence of change set not a period but a criterion. Above the fifth policy means that each versions and change sets are stored using a criterion. Scheme 5 stores the versions created when the most update operations within a period are occurred. The criterion is determined based on a number of update operations. This scheme can reduce the version creation time when temporal queries are requested.

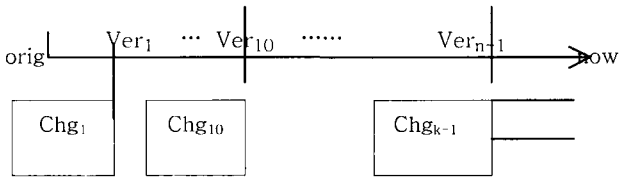


Fig. 2 Scheme of storing all the change sets.

Fig.2 shows the storage policy concept of storing all the change sets. The originated version and the last version (current version) are stored in the storage. Also, A change set is stored whenever a version is generated. However, Versions those are between the originated and the current version, they are not stored. In Figure2, Chg_i means a set of change. A version is represented by an entity: (Ver_{i-1}, Chg_i). $Ver_{i-1} \circ Chg_i$ generate Ver_i, i.e., Ver_i is generated by applying Chg_i to Ver_{i-1}. If only generating versions must start from the originated version. Here origin is a fixed time specifying the creation time of the particular ontology and now corresponds to the current time.

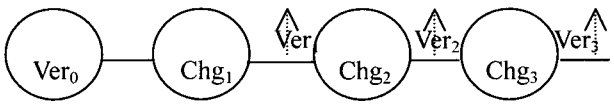


Fig. 3 Stored versions and change sets in Scheme 3.

Sequence of stored versions is shown in Figure 3. Scheme 3 stores the first version, the last version and just change sets. Applying Chg_1 to Ver_0 generates Ver_1, i.e., $Ver_1 = Ver_0 \circ Chg_1$. Also, We can get Ver_2 applying Chg_2 to result of $Ver_0 \circ Chg_1$, i.e., $Ver_2 = (Ver_0 \circ Chg_1) \circ Chg_2$.

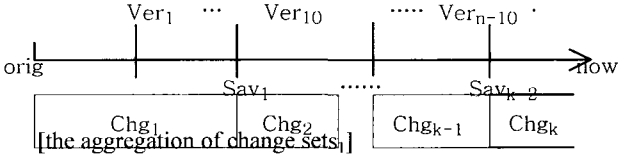


Fig. 4 Scheme of storing the versions the aggregation of change sets periodically.

Above we mentioned scheme 4 that is the versions and the aggregation of change sets periodically, in this scheme, we store a particular versions and an aggregation of changes sets periodically in the repository. Figure 4 shows scheme of this storage policy. The storing period can become the time granularity. At this scheme we use an aggregation of change sets, which means to group by the granularity. We have a 4-level time hierarchy: year, month, week, and day. Here we have days at the finest granularity. Change sets can be grouped in days, weeks, months, or years, and then stored in the repository. In this figure, we assumed the storing period is 10 that is constant. Here Ver_1 is virtual version that is not stored physically and Ver_{10} is stored version. The Chg_1 is the aggregation of change sets that aggregates all change sets before Ver_{10} from origin to Sav_1. The versioned data Ver_{10} and the aggregation of change sets Chg_1 are stored in the repository at point with time Sav_1. We assume that Ver_{n-2} is not stored physically as virtual version, and then Ver_{n-2} is reconstructed with the aggregation of change sets before Ver_{n-1} from time Sav_{k-2} to time Sav_{k-1}.

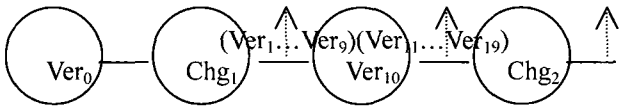


Fig. 5 Stored versions and change sets in Scheme 4.

Scheme 4 stores versions and change sets as shown in Fig. 5 All versions between Ver_1 and Ver_9 are generated by applying Chg_1 to Ver_0. Also, applying Chg_2 to Ver_{10} generates all versions between Ver_{11} and Ver_{19}. To generate a particular version, a change set is applied to it's a previous version.

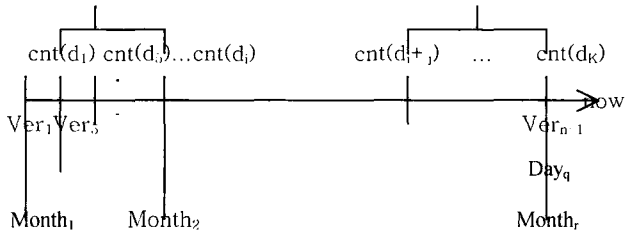


Fig. 6 Scheme of storing the versions and the aggregation of change sets using a criterion.

The storage policy that is storing the versions and the aggregation of change sets using a criterion is shown in Figure 6. The basic idea of this storage policy is to get maximum number of update operations. The maximum value corresponds to specific time, which is a criterion to store a version. To get the criterion, we execute the count function a day during a month and get the maximum value from all the count value. Here *cnt* is a count function to get a number of update operations and $crit_i$ is a maximum value corresponding to specific time. If $cnt(d_5)$ is a maximum value in this figure, $crit_1$ becomes a criterion to store a particular version Ver_5. Just a version is stored at a time instant corresponding a criterion. Other versions are not stored physically during time granularity except a version that corresponds to a criterion; only aggregations of change sets are stored.

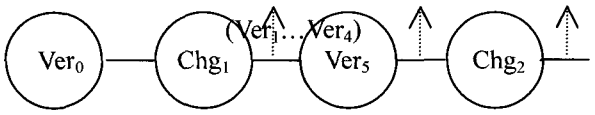


Fig. 7 Stored versions and change sets in Scheme 5.

Figure 7 shows stored versions and change sets in Scheme 5. In order to reduce the version generation time, we need to get a version that has a maximum value of update operations during time granularity. For example, Ver_5 has a maximum value of update operations. Ver_5 is already stored in repository, we don't need to generate it. A lot of processing time to need for generating Ver_5 is saved.

Algorithm 1 Storing the versions and the aggregation of change sets periodically

1. Decide time granularity *granul* for storing change sets.
2. For the nearest time Sav_i creating version from specific period *granul*.
3. Store change sets Chg_i from Sav_{i-1} to Sav_i.
4. Store versions created at the time Sav_i.

Algorithm 2 Storing the versions and the aggregation of change sets using a criterion

1. Decide time granularity *granul* for storing change sets.
2. When current time is the time corresponding to *granul*.
3. Count number of update operations for each version generation time.
4. Get a maximum value as criterion for above No.3

Mentioned above two storage policies, Scheme 4 and 5, respectively, the algorithm for storing the versions and the aggregation of change sets is presented in Algorithm 1 and 2.

V. PERFORMANCE EVALUATIONS

In this section, we present the experimental results for evaluating the performance of the five storage policies described in Section 4. We compare the storage space usage and the average access time of the five schemes. We use the following experimental settings:

- A size of 100Megabytes for ontology
- A lifespan of 365 for simulation interval
- 50% current queries, 50 % past queries

Figure 8 shows the storage space usage according to changing change set size. The size of change set varies from 5% to 45% of the ontology size. We can see that Scheme 3 has the least storage usage. This is due to the fact that Scheme 3 just stored change sets between the first version and the last version. However, we can see that Scheme 1 consumes the largest space because all versions are stored. Scheme 4 and Scheme 5 have similar

storage space usage, because they store versions and change sets in turn. Difference of the storage space usage among Scheme 3, Scheme 4, and Scheme 5 is small. The Scheme 4 and the Scheme 5 uses a little more space than the Scheme 3.

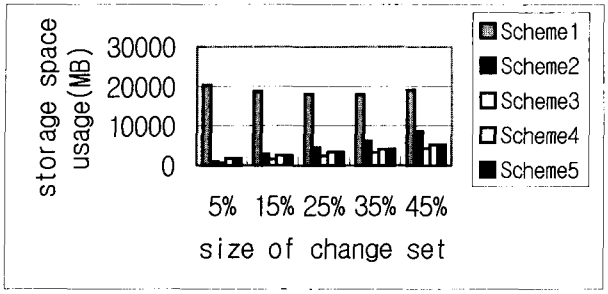


Fig. 8 Storage space usage.

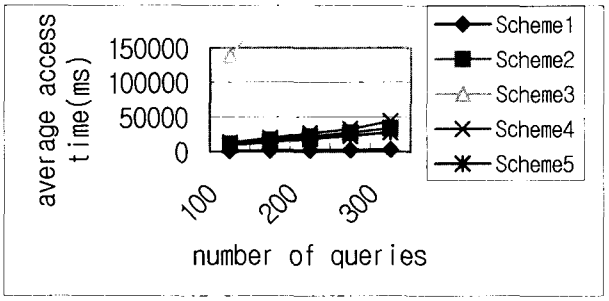


Fig. 9 Average access time.

Figure 9 shows the average access time for the five schemes. We can see that Scheme 3 has the largest average access time and Scheme 1 has the smallest one. The average access time in Scheme 1 increases rapidly as the number of queries increases. This is due to that Scheme 3 needs very much versions creation time because it just stored change sets. On the other hand, Scheme 1 does not need versions creation time because it stored all the versions. In Figure 8 and Figure 9, we can see that Scheme 5 has less storage space usage and less access time. Scheme 5 is a steady storage policy on both space and time sides.

VI. CONCLUSIONS

In this paper we described about the management of versions in ontology. We have studied a set of changes based on domain changes, changes in conceptualization, metadata changes, and temporal dimension. Our change specification is represented by a set of changes. A set of changes consists of instance data change, structural change, identifier change, meta data change, and temporal dimensions. In order to support an ontology query language that supports temporal operations, we considered temporal dimension includes transaction time and valid time. Ontology versioning brings about massive amount of versions to be stored and maintained. We discussed several storage policies of versioned data: Storing all the versions, Storing all the sequence of changed elements, Storing all the change sets, Storing the versions and the aggregation of change sets periodically, and Storing the versions and the aggregation of change sets using a criterion. Also, we presented the experimental results for evaluating the performance of different storage policies from scheme 1 to scheme 5. More experiments are planned for the near future.

REFERENCES

[1] T. Berners-Lee (with Mark Fischetti), Weaving the Web, The original design and ultimate destiny of the World Wide Web, Harper, 1999.

[2] T. Berners-Lee, J. Hendler, and O. Lassila, The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities, Scientific American, May 2001.

[3] N. Guarino, "Formal Ontology in Information Systems," Proc. of the 1st International Conference, Trento, Italy, 6-8 June 1998.

[4] M. Klein and D. Fensel, "Ontology versioning for the Semantic Web," In Proceedings of the International Semantic Web Working Symposium (SWWS), pages 75-91, Stanford University, California, USA, 2001.

[5] N. F. Noy, "Ontology Engineering," In Proceedings of the International Semantic Web Working Symposium (SWWS), page 2, Stanford University, California, USA.

[6] T. R. Gruber, "A translation approach to portable ontology specifications," Knowledge Acquisition, 5(2), 1993.

[7] N. F. Noy and M. Klein. "Ontology evolution: Not the same as schema evolution," Knowledge and Information Systems, 5, 2003. in press.

[8] N. Noy and M. Musen, "PROMPTDIFF: A fixed-point algorithm for comparing ontology versions," In 18th National Conference on Artificial Intelligence (AAAI2002), 2002.

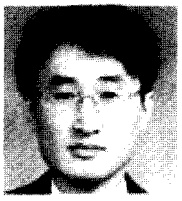
[9] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. P. Rice, "OKBC: A programmatic foundation for knowledge base interoperability," In 15th Nat. Conf. on Artificial Intelligence (AAAI-98), pages 600–607, 1998.

[10] G. Wiederhold, "An Algebra for Ontology Composition," In Proceedings of 1994 Monterey Workshop on Formal Methods, pages 56-62. U.S. Naval Postgraduate School, 1994.

[11] M. Klein, A. Kiryakov, D. Ognyanov, and D. Fensel, "Ontology Versioning and Change Detection on the Web," In 13th International Conference on Knowledge Engineering and Knowledge Management *(EKAW02)*, Sigüenza, Spain, October 1-4, 2002.

[12] C. S. Jensen, C. E. Dyreson, (Eds.), M. B¨ohlen, J. Clifford, R. A. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. K. Jajodia, W. K¨afer, N. Kline, N. Lorentzos, Y. Mitsoupoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. U. Tansel, P. Tiberio, G. Wiederhold, The Consensus Glossary of Temporal Database Concepts - February 1998 Version, in: O. Etzion, S. Jajodia, S. Sripada (Eds.), Temporal Databases—Research and Practice, Springer-Verlag, 1998, pages 367–405, INCS No. 1399.

[13] J. Heftin and J. A. Hendler, "Dynamic ontologies on the web," In Proc. of AAAI/IAAI 2000, pages 443-449, 2000.
[14] A. Marian, S. Abiteboul, G. Cobena, L. Mignet, "Change-Centric Management of Versions in an XML Warehouse," In Proc. of 27th Int. Conf. on Very Large Data Bases (VLDB). pages 581–590, 2001.
[15] S. Y. Chien, V. J. Tsotras, C. Zaniolo, "XML Document Versioning," SIGMOD Record 30, pages 46 – 53, 2001.
[16] B. Benatallah, M. Mahdavi, P. Nguyen, Q. Z. Sheng, L. Port, B. McIver, "An Adaptive Document Version Management Scheme," The 15th Conference on Advanced Information Systems Engineering (CAiSE'03), pages 16-20, Austria, 2003.
[17] A. Maedche, B. Motik, L. Stojanovic, R. Studer, R. Volz, "An Infrastructure for Searching, Reusing and Evolving Distributed Ontologies," WWW2003, Hungary, 2003.
[18] L. Deborah, McGuinness, R. Fikes, J. Rice, S. Wilder. "An Environment for Merging and Testing Large Ontologies," In Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000), Colorado, 2000.



Hong-Won Yun
Received his B. S. and the Ph.D. degrees in Department of Computer Science from Pusan National University, Pusan, Korea, in 1986 and 1998, respectively. From 1996 to now, he is an Associate Professor, Division of Computer & Information Engineering, Silla University in Korea. His research interests include Semantic Web, Temporal Database.