key of the table represented by E_T. For theoretical treatment of aggregates in XDD theory, the reader is referred to [2, 5].

6 Conclusions

As demonstrated in this paper, mapping rules for transforming UML class diagrams to relational database schemas can be represented using XML definite clauses. A prototype UML knowledge-based system under the framework outlined in the first section has been developed and satisfactory results have been obtained. Since XMI is becoming a standard textual representation of UML diagrams, it is expected that the presented framework has several other promising applications, such as reverse and forward engineering UML models and consistency verification of models. As virtually every tool supporting UML is capable of reading and writing models using XMI, integration of the presented knowledge-based approach into other UML-based software modeling techniques is possible.

Acknowledgement

This work was supported by the Thailand Research Fund, under Grant No. PDF/31/2543.

References

[1] Akama, K., *Declarative Semantics of Logic Programs on Parameterized Representation Systems*, Advances in Software Science and Technology, **5**, 45–63, 1993.

[2] Akama, K., Anutariya, C., Wuwongse, V., and Nantajeewarawat, E., *A Foundation for XML Document Databases: Query Formulation and Evaluation*, Technical Report, CSIM, Asian Institute of Technology, Thailand, 1999.

[3] Akama, K., Shimitsu, T., and Miyamoto, E., *Solving Problems by Equivalent Transformation of Declarative Programs*, J. Japanese Society of Artificial Intelligence, **13**(6), 944–952, 1998.

[4] Akama, K., Shigeta, Y., and Miyamoto, E., *Solving Problems by Equivalent Transformation of Logic Programs*, Proc. 5th Intl. Conf. on Information Systems Analysis and Synthesis, Orlando, Florida, 1999.

[5] Anutariya, C., Wuwongse, V., Nantajeewarawat, E., and Akama, K., *Towards a Foundation for XML Document Databases*, Proc. Intl. Conf. on E-Commerce and Web Technologies, London-Greenwich, UK, Lecture Notes in Computer Science, Vol. 1875, pp. 324–333, Springer-Verlag, 2000.

[6] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley, 1998.

[7] Brown, K. and Whitenack, B. G., *Crossing Chasms: A Pattern Language for Object-RDBMS Integration*, J. Vlissides et. al. (eds.), *Pattern Languages of Program Design 2*, ch.14, Addison-Wesley, 1996.

[8] Demuth, B. and Hussmann, H., *Using UML/OCL Constraints for Relational Database Design*, Proc. 2nd Intl. Conf. on the Unified Modeling Language, Lecture Notes in Computer Science, Vol. 1723, pp. 598–613. 1999.

[9] Goldfarb, C. F. and Prescod, P., *The XML Handbook*, Prentice Hall, 1998.

[10] Keller, W., *Mapping Objects to Tables*, Proc. 1997 European Pattern Languages of Programming Conf., Irrsec, Germany, 1997.

[11] Nantajeewarawat, E., Wuwongse, V., Anutariya, C., Akama, K., and Thiemjarus, S., *Towards Reasoning with UML Diagrams Based-on XML Declarative Description Theory*, Proc. Intl. Conf. on Intelligent Technologies, Bangkok, Thailand, pp. 341–350, 2000.

[12] Nantajeewarawat, E. and Wuwongse, V., *Defeasible Inheritance Through Specialization*, Computational Intelligence, **17**(1), 62–86, 2001.

[13] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[14] Wuwongse, V., Anutariya, C., Akama, K., and Natajeewarawat, E., *XML Declarative Description: A Language for the Semantic Web*, IEEE Intelligent Systems, **16**(3), 54–65, 2001.

[15] Wuwongse, V. and Nantajeewarawat, E., *Declarative Programs with Implicit Implication*, IEEE Transactions on Knowledge and Data Engineering. (To appear)

[16] *XML Metadata Interchange Format (XMI)*, IBM Application Development, www-4.ibm.com/software/ad/standards/xmi.html.

Appendix

As Equivalent Transformation Interpreter (ETI), the inference engine used in the current prototype implementation of this work, operates on facts that are encoded in the form of s-expressions, XMI representations of UML diagrams are converted into data of this form. Such conversion is straightforward, and can directly be implemented through the mapping shown in Figure 17. Furthermore, in order to make inferences from the obtained s-expressions according

XMI Representation	S-Expression Representation
$<tag\ attr_1 = val_1\ \ldots\ attr_n = val_n/>$	$((tag\ (attr_1\ val_1)\ldots(attr_n\ val_n)))$
$<tag\ attr_1 = val_1\ \ldots\ attr_n = val_n> val </tag>$	$((tag\ (attr_1\ val_1)\ldots(attr_n\ val_n)(\texttt{content}\ val)))$
$<tag\ attr_1 = val_1\ \ldots\ attr_n = val_n>$ *subElements* $</tag>$	$((tag\ (attr_1\ val_1)\ldots(attr_n\ val_n))\ S)$ where S is one or more s-expression(s) representing the XML element(s) *subElements*

Figure 17: Mapping from XMI representations to S-expressions, where *tag* is a tag name, the $attr_i$ are attribute names, *val* and the val_i are strings, and *subElements* is one or more XML element(s)

```
0  /* The ET rule prepared from the clause C_TN1 */
1  (Rule TN1-GenTableNm
2   (Head (TABLENAME *TN))
3   (Body (exec (= *TN
4                  ((TableName (idref *CId)
5                              (name *Nm)))))
6         (FACT *X)
7         (member ((*Tag | *pairs) | ?)  *X)
8         (member (xmi.id *CId) *pairs)
9         (member (name *Nm) *pairs)
10        (member *Tag
11                (UML:Class UML:AssociationClass))
12  ))
13 /* The ET rule prepared from the clause C_CL1 */
14 (Rule CL1-GenColumn
15  (Head (COLUMN *COL))
16  (Body (exec (= *COL
17                 ((Column (idref *CId) (name *ANm)
18                          (type *TNm)))))
19        (FACT *X)
20        (member ((*Tag | *pairs1) | *EVAR1) *X)
21        (member (xmi.id *CId) *pairs1)
22        (+ ?  (((UML:Attribute | *pairs2) | ?))
23              ((*Tag | *pairs1) | *EVAR1))
24        (member (name *ANm) *pairs2)
25        (member (type *AType) *pairs2)
26        (FACT ((UML:DataType | *pairs3) | ?))
27        (member (xmi.id *AType) *pairs3)
28        (member (name *TNm) *pairs3)
29        (member *Tag
30                (UML:Class UML:AssociationClass))
31  ))
```

Figure 18: Examples of ET rules

to the specifications provided by the XML definite clauses presented in Section 5, a set of procedural rewriting rules, called Equivalent Transformation (ET) rules, will be prepared from the clauses. These ET rules together with the control mechanism of ETI specify a backward-chaining-like procedure for generating tables and their components. As illustrative examples, the ET rules prepared from the clauses C_{TN1} in Figure 8 and C_{CL1} in Figure 9 are shown in Figure 18 (Lines 1–12 and 14–31, respectively), where a term beginning with the asterisk is regarded as a variable. Each of these two ET rules consists of two parts: `Head` part and `Body` part. The `Head` part of a rule specifies the pattern of expressions to which the rule is applicable; i.e., the rule is only applicable to an expression that is more specific than the specified pattern. When applied, the rule transforms an expression (which is given

```
((Table (idref "S.7") (name "Student"))
   ((Column (name "ID") (isprimary "true")
            (isunique "true") (isnull "false")
            (type "Integer")))
   ((Column (name "studentID") (type "String")))
   ((Column (name "accGPA") (type "Double")))
   ((Column (name "major") (type "String")))
   ((Column (name "creditsEarned")
            (type "Integer")))
   ((Column (name "staysAt") (type "Integer")
            (reference "Residence")
            (isnull "true")))
   ((Column (name "PersonID")
            (type "Integer") (isunique "true")
            (reference "Person"))))
```

Figure 19: A derived s-expression

as a goal) into zero or more expression(s) (which are then regarded as new goals) of the pattern(s) specified in its `Body` part. (In general, an ET rule may also contain some additional conditions for determining its applicability.)

Consider, for instance, the clause C_{TN1} in Figure 8 and the first ET rule in Figure 18. This ET rule specifies the transformation procedure for any goal s-expression representing a `TableName`-element. When the rule is applied, such a goal s-expression will be unified with the s-expression specified in Lines 4–5 and replaced with the five s-expressions specified in Lines 6–11. After the `Fact`-expression (Line 6) is processed (by some other rule), the variable `*X` will be instantiated into an s-expression representing some diagram component, say D, from which certain information will be extracted and tested according to the specification given by the clause C_{TN1}. The three `member`-expressions in Lines 7–9 are used to extract the tag name and the values of the attributes `xmi.id` and `name` of the XML element representing the diagram component D; then, the `member`-expression in Lines 10–11 tests whether the extracted tag name is `UML:Class` or `UML:AssociationClass`.

Figure 19 illustrates an s-expression obtained from the prototype system when it is tested with the class diagram in Figure 2. This s-expression represents the generated table for the class Student in the diagram. (The `idref`-expression enclosed in each `Column`-expression is omitted.)

Published in Tanprasert, T., editor, Proceedings of the Second International Conference on Intelligent Technologies (InTech'2001), Bangkok, Thailand, pages 392-401, November 2001. ISBN 974-615-068-5.

Expanding Transformation: A Basis for Verifying the Correctness of Rewriting Rules

Ekawit Nantajeewarawat
IT Program
Sirindhorn Intl. Inst. of Tech.
Thammasat University
Pathumthani 12121, Thailand
E-mail: ekawit@siit.tu.ac.th

Kiyoshi Akama
Center for Information and Multimedia Studies
Hokkaido University
Sapporo 060-0811, Japan
E-mail: akama@cims.hokudai.ac.jp

Hidekatsu Koike
Div. of System & Info. Eng.
Faculty of Engineering
Hokkaido University
Sapporo 060-0811, Japan
E-mail: koke@cims.hokudai.ac.jp

Abstract: Unfolding transformation is considered as the composition of two simpler operations, i.e., expanding transformation and unification. Then it is pointed out that expanding transformation rather than unfolding transformation serves as a suitable basis for verifying the correctness of rewriting rules by means of pattern manipulation, which in turn is an underlying mechanism for systematically generating rewriting rules from a problem description. The correctness of expanding transformation is established. The correctness of a basic class of rewriting rules, called general rewriting rules, is shown thereupon. The application of expanding transformation and the correctness thereof to the correctness verification of a larger class of rewriting rules, called expanding-based rewriting rules, by transformation of clause patterns is demonstrated.

Key words: Rule-based equivalent transformation, Rewriting rules, Pattern matching, Expanding transformation, Unfolding, Semantics preservation, Rule-based systems, Declarative descriptions

1 Introduction

As a fundamental transformation rule, the unfolding rule has long been used in the context of functional programs for the computation of recursively defined functions and for developing recursive equation programs [8]. The rule consists in replacing an instance of the left-hand side of a recursive equation by the corresponding instance of the right-hand side. By taking its application, which can be regarded as a symbolic computation step, to be equivalent to an application of the resolution inference rule [12], the unfolding rule has been adapted to the case of logic programs [11, 15].

Although the unfolding rule for logic programs is derived directly from that used in functional programming, there is a remarkable contrast between their application. In the case of logic programs, to unfold a definite clause with respect to a body atom B using a set of definite clauses P, B need not be an instance of the head of some clause in P—it is only required that B is unifiable with the head of some clause in P. This discrepancy stems from the fact that a *unifying* substitution is used in a resolution step—not a *pattern-matching* substitution, which is used in a replacement step for functional programs.

It has been argued in [2, 4, 5] that instead of basing computation solely upon the resolution inference rule and the fixed procedural interpretation of definite clauses (as it happens in the logic programming paradigm [9]), more efficient and effective computation can be obtained through semantics-preserving transformation of a set of definite clauses by applying user-definable rewriting rules in a user-controllable way. This conviction brought about a new promising computation framework, called *equivalent transformation (ET)* framework [5, 6], which has provided a solid foundation for knowledge processing systems in several application domains [7, 10, 13, 14, 16].

In the ET paradigm, the applicability of a rewriting rule is determined by pattern matching rather than unification; as a result, a rewriting rule can be tailored for some specific pattern of atoms for improvement of computation efficiency. It will be demonstrated in this paper that a more basic kind of transformation—called *expanding transformation*—rather than unfolding transformation provides a suitable basis for discussing the correctness and application of an important class of rewriting rules by manipulation of patterns of atoms and patterns of clauses.

Such pattern manipulation in turn forms a basis for meta-level computation for automatic generation of rewriting rules from a set of definite clauses in the framework proposed in [3]. The purpose of this paper is threefold:

- The concept of expanding transformation will be introduced; its correctness will be proved based on an appropriate formulation of the meanings of declarative descriptions.
- A theoretical basis for justifying the correctness of a rewriting rule will be established. A basic class of rewriting rules, called *general rewriting rules*, will be defined. Their correctness will be proved through the correctness of expanding transformation.
- A larger class of rewriting rules, called *expanding-based rewriting rules*, will be introduced. Based on manipulation of atom patterns and clause patterns, the application of expanding transformation to the correctness verification of rewriting rules in this class will be illustrated.

By decomposition of an unfolding step into an expanding step and a unification step, Section 2 provides an informal introduction to expanding transformation; then, it explains the appropriateness of expanding transformation as a foundation for discussing the correctness of rewriting rules based on pattern manipulation. Section 3 defines preliminary syntactic components, which are used for defining declarative descriptions and their meanings in Section 4, and rewriting rules, their application, and their correctness in Section 6. Section 5 formally defines expanding transformation and proves its correctness, which is then used for verifying the correctness of general rewriting rules and expanding-based rewriting rules in Sections 7 and 8, respectively.

2 Motivation

In the ET model, a problem is formulated as a *declarative description*, represented by the union of two sets of definite clauses, one of which is called the *definition part*, and the other the *query part*. The definition part provides general knowledge about the problem domain and describes some specific problem instances. The query part specifies a question regarding the content of the definition part. The problem is solved by transforming the query part successively, based on the definition part, into a simpler but equivalent set of definite clauses from which the answers to the specified question can be obtained directly.

2.1 Unfolding = Expanding + Unification

Consider a simple problem formulated as the union of a definition part D_{ap} consisting of the three definite clauses

C_{ap_1}: $app([\,], Y, Y) \leftarrow$
C_{ap_2}: $app([A|X], Y, [A|Z]) \leftarrow app(X, Y, Z)$
C_{eq}: $eq(X, X) \leftarrow$

(where *app* and *eq* stand for *append* and *equal*, respectively) and a query part Q_1 containing only the definite clause

C_1: $answer(X', Y') \leftarrow app(X', [Y'], [7])$.

As the *app*-atom in the body of C_1 is unifiable with the head of C_{ap_1}, using the unifying substitution $\theta_1 = \{X'/[\,], Y/[7], Y'/7\}$, and with the head of C_{ap_2}, using the unifying substitution $\theta_2 = \{X'/[7|X], Y/[Y'], A/7, Z/[\,]\}$, the query part Q_1 can be transformed by unfolding C_1 using D_{ap} into a new query part Q_2 consisting of the two definite clauses

C_2: $answer([\,], 7) \leftarrow$
C_3: $answer([7|X], Y') \leftarrow app(X, [Y'], [\,])$.

From C_2 and θ_1, an answer to the query part Q_1, i.e., $X' = [\,]$ and $Y' = 7$, can be obtained effortlessly. Notice that since the body atom of C_3 is not unifiable with the head of any clause in D_{ap}, no clause can be obtained by unfolding C_3 using D_{ap}; as a result, there is no other answer to the query part.

An unfolding step can be considered as the composition of two successive more elementary computation steps: an *expanding* step and a *unification* step. For instance, the unfolding step transforming Q_1 into Q_2 can be decomposed as follows. First, expand C_1 using D_{ap}—that is, for each clause C in D_{ap} whose head is an *app*-atom, rewrite C_1 into another clause by simply replacing the *app*-atom in the body of C_1 with the body of C along with three *eq*-atoms equalizing the arguments of the replaced *app*-atom and the corresponding arguments of the head of C. This expanding step transforms Q_1 into a set Q_2' comprising the two clauses

C_2': $answer(X', Y') \leftarrow eq(X', [\,]), eq([Y'], Y), eq([7], Y)$
C_3': $answer(X', Y') \leftarrow eq(X', [A|X]), eq([Y'], Y), eq([7], [A|Z]), app(X, Y, Z)$.

Next, unify the arguments of each *eq*-atom in C_2' as well as those of each *eq*-atom in C_3'; thereby, the clauses C_2 and C_3 are obtained. The formal definition of expanding transformation will be given in Section 5.

2.2 Pattern Matching and Rewriting Rules

Instead of using the unfolding rule, one may devise a rewriting rule for transforming atoms of some specific pattern. From the definition part D_{ap}, for example, one may specify as a rule that an *app*-atom whose second and third arguments are both (possibly non-ground) singleton lists, say L and L', can be removed from the body of a clause by

- equalizing the first argument of the *app*-atom and the empty list, and
- equalizing the element of L and that of L'.

Using this rule, the query part Q_1 of Subsection 2.1 can be transformed in one step into a query part Q_3 containing only the clause

$$C_4: \quad answer(X', Y') \leftarrow eq(X', []), eq(Y', 7).$$

This transformation step can be described more precisely by the rewriting rule

$$r_1: \quad app(\&X, [\&Y], [\&Z]) \rightarrow eq(\&X, []), eq(\&Y, \&Z),$$

where the arrow "$\rightarrow$" intuitively means "can be replaced with" and the left-hand side and the right-hand side of r_1 specify the pattern of atoms to which the rule is applicable and the pattern of replacement atoms, respectively. By instantiating $\&X, \&Y$ and $\&Z$, which will be referred to as *meta-variables*, into the terms X', Y' and 7, respectively, the pattern in the left-hand side matches the body atom of C_1 and that in the right-hand side is instantiated into the body atoms of C_4—that is, by applying r_1 to the body atom of C_1 using this instantiation, C_1 is transformed into C_4. Determination of rule applicability by pattern matching, as opposed to unification, makes the rule r_1 applicable only to atoms of the desired pattern. The syntax for rewriting rules as well as their application will be precisely described in Section 6.

By the application of r_1, not only does the resulting clause C_4 in Q_3 directly yield an answer ($X' = []$ and $Y' = 7$) to the query part Q_1; in addition, the absence of any other clause in Q_3 indicates immediately that there is no other answer. In comparison, from the set $Q_2 = \{C_2, C_3\}$ obtained by the application of the unfolding rule in the preceding subsection, some further computation is required in order to find that no clause can be derived by further unfolding C_3 using D_{ap}, and no other answer exists. In particular, if the body of C_3 additionally contains some atom that is unifiable with the head of some clause in D_{ap}, then several useless further unfolding steps may take place. It is demonstrated in [5] that, in general, computation efficiency can be significantly improved by avoiding transformation steps that increase the number of clauses.

In the ET paradigm, rewriting rules will be prepared from a given definition part, and a set of prepared rewriting rules, instead of the definition part itself, will be regarded as a program. Based on meta-level manipulation of atom patterns, a method for systematically generating rewriting rules from a definition part is developed in [3].

2.3 Expanding Transformation as a Basis for Meta-Level Transformation

Expanding transformation and transformation by application of rewriting rules based on pattern matching have a common characteristic, i.e., they do not use unification—consequently, they do not instantiate any variable occurring in a replaced atom. Considering the body atom $app(X', [Y'], [7])$ of C_1 and the transformation steps in Subsections 2.1 and 2.2, for example, while X' is instantiated into $[]$ and $[7|X]$ by unfolding C_1 into C_2 and C_3, neither X' nor Y' is instantiated by expanding C_1 into C_2' and C_3', and neither of them is instantiated by rewriting C_1 using the rule r_1 into C_4.

In the framework for generating rewriting rules by means of mata-computation—by manipulation of patterns of atoms rather than ordinary atoms—proposed in [3], meta-variables such as $\&X$, $\&Y$ and $\&Z$ are used to represent arbitrary ordinary terms. As a representative of all terms, a meta-variable of this kind should not be instantiated into any specific term in a pattern-manipulation process. Accordingly, expanding transformation provides a befitting basis for discussing the correctness of rewriting rules in this framework. As an illustrative example, the correctness of the rewriting rule r_1 of Subsection 2.2 can be justified by transformation of clause patterns as follows. From a clause of the form

$$\hat{C}_1: \quad \hat{H} \leftarrow \ldots, app(\&X, [\&Y], [\&Z]), \ldots,$$

where $\hat{H}$ represents an atom of any arbitrary pattern and the meta-variables $\&X, \&Y$ and $\&Z$ represent any arbitrary terms, one can expand $\hat{C}_1$ using D_{ap} into

$$\hat{C}_2: \quad \hat{H} \leftarrow \ldots, eq(\&X, []), eq([\&Y], Y), eq([\&Z], Y), \ldots$$

$$\hat{C}_3: \quad \hat{H} \leftarrow \ldots, eq(\&X, [A|X]), eq([\&Y], Y), eq([\&Z], [A|Z]), app(X, Y, Z), \ldots .$$

Then, $\hat{C}_3$ can be further expanded using D_{ap} into

$\hat{C}_3'$: $\hat{H} \leftarrow \ldots, eq(\&X, [A|X]), eq([\&Y], Y),$
$eq([\&Z], [A|Z]),$
$eq(X, []), eq(Y, Y1),$
$eq(Z, Y1), \ldots$

$\hat{C}_3''$: $\hat{H} \leftarrow \ldots, eq(\&X, [A|X]), eq([\&Y], Y),$
$eq([\&Z], [A|Z]),$
$eq(X, [A1|X1]), eq(Y, Y1),$
$eq(Z, [A1|Z1]),$
$app(X1, Y1, Z1), \ldots$,

both of which can be deleted by constraint solving for *eq*-atoms (for example, $\hat{C}_3''$ can be removed since any ground instantiation equalizing simultaneously the two arguments of each of its *eq*-atom necessarily instantiates Z into the empty list and, at the same time, a non-empty list, which is impossible whatever terms the meta-variables $\&X, \&Y$ and $\&Z$ represent). Next, by simplifying its body, $\hat{C}_2$ can be rewritten into

$\hat{C}_2'$: $\hat{H} \leftarrow \ldots, eq(\&X, []), eq([\&Y], [\&Z]), \ldots$,

which can be further simplified into

$\hat{C}_2''$: $\hat{H} \leftarrow \ldots, eq(\&X, []), eq(\&Y, \&Z), \ldots$.

This means a clause containing any atom B of the pattern $app(\&X, [\&Y], [\&Z])$ can be transformed into another clause by replacing B with its corresponding atoms of the patterns $eq(\&X, [])$ and $eq(\&Y, \&Z)$; thus, the rule r_1 is correct [3, 4, 5].

It is important to note that in general unfolding cannot be employed in such manipulation of atom patterns. For instance, to unfold $\hat{C}_1$ with respect to $app(\&X, [\&Y], [\&Z])$ using D_{ap}, the meta-variable $\&X$ has to be unified with $[]$, which is only possible when $\&X$ represents a variable or the empty list. As a result, in the presence of a meta-variable representing any arbitrary term, unfolding transformation is usually not applicable.

3 Basic Syntactic Components

After specifying the alphabet used in the paper, some basic concepts, e.g., terms and atoms, along with the concepts of meta-term and meta-atom, which are used for specifying patterns of terms and atoms, respectively, will be defined.

Alphabet An *&-variable* is a variable that begins with the symbol &; e.g., $\&N$ and $\&X$ are &-variables. A *#-variable* is a variable that begins with the symbol #; e.g., $\#X$ and $\#Y$ are #-variables. An &-variable as well as a #-variable is called a *meta-variable*. &-variables and #-variables have different instantiation characteristics, which will be rigorously specified in Section 6. An alphabet $\Delta = \langle K, F, V, R\rangle$ is assumed, where K is a set of constants, including integers and *nil*; F a set of functions, including the binary function *cons*; V is the disjoint union of a set V_1 of ordinary variables and a set V_2 of meta-variables; and R is the union of two mutually disjoint sets of predicates $R_1 = \{app, eq, \ldots\}$ and $R_2 = \{answer, \ldots\}$. An ordinary variable in V_1 is assumed to begin with neither & nor #. When no confusion is possible, an ordinary variable in V_1 and a meta-variable in V_2 will be simply called a variable and a meta-variable, respectively.

Terms, Meta-Terms, Atoms, Meta-Atoms, and Substitutions Usual first-order terms on $\langle K, F, V_1\rangle$ and on $\langle K, F, V_2\rangle$ will be referred to as *terms* and *meta-terms*, respectively, on Δ. Given $R' \subseteq R$, usual first-order atoms on $\langle K, F, V_1, R'\rangle$ and on $\langle K, F, V_2, R'\rangle$ will be referred to as *atoms on* R' and *meta-atoms on* R', respectively. The standard Prolog notation for lists is adopted; e.g., $[X, Y]$ and $[7, \#X|\&Y]$ are abbreviations for the term $cons(X, cons(Y, nil))$ and the meta-term $cons(7, cons(\#X, \&Y))$, respectively. First-order atoms on $\langle K, F, \emptyset, R\rangle$ are called *ground atoms* on Δ. In the sequel, let $\mathcal{T}_\Delta$ be the set of all terms on Δ, and $\mathcal{G}_\Delta$ the set of all ground atoms on Δ; also let $\mathcal{A}_i$ and $\hat{\mathcal{A}}_i$ be the set of all atoms and the set of all meta-atoms, respectively, on R_i, where $i \in \{1, 2\}$. A *substitution* on Δ is a set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i belongs to V_1, each t_i is a term on Δ such that $v_i \neq t_i$, and the v_i are all distinct. Each v_i/t_i is called a *binding* for v_i. Let $\mathcal{S}_\Delta$ be the set of all substitutions on Δ. A substitution $\theta \in \mathcal{S}_\Delta$ is called a *variable-renaming* substitution, if and only if for any binding v/t in θ, $t \in V_1$ and for any other binding v'/t' in θ, $t \neq t'$.

4 Declarative Descriptions and Their Meanings

In general, the ET model can deal with several data structures other than usual first-order terms, e.g., multisets and XML data, and the concept of declarative description can be extended with these data structures [1, 16]. For simplicity, however, only usual terms are used in this paper. Subsection 4.1 specifies the forms of definite clauses and declarative descriptions discussed herein; Subsection 4.2 provides some basic concepts used for defining the meanings of declarative descriptions in Subsection 4.3 and their related results used for verifying the correctness of expanding transformation in Section 5.

4.1 Declarative Descriptions

A *definite clause* C *on* Δ is an expression of the form $A \leftarrow Bs$, where A is an atom on R and Bs is a (possibly empty) set of atoms on R. The atom A is called the *head* of C, denoted by $head(C)$; the set Bs is called the *body* of C, denoted by $Body(C)$; each element of $Body(C)$ is called a *body atom* of C. When $Body(C) = \emptyset$, C will be called a *unit clause*. The set notation is used in the right-hand side of C so as to stress that the order of the atoms in $Body(C)$ is immaterial. However, for the sake of simplicity, the braces enclosing the body atoms in the right-hand side of a definite clause will often be omitted; e.g., a definite clause $A \leftarrow \{B_1, \ldots, B_n\}$ will often be written as $A \leftarrow B_1, \ldots, B_n$.

Let $R' \subseteq R$. A definite clause C is said to be *from* R_1 *to* R', if and only if each element of the body of C is an atom on R_1 and the head of C is an atom on R'. A *declarative description from* R_1 *to* R' is a set of definite clauses from R_1 to R'. The set of all declarative descriptions from R_1 to R' will be denoted by $Dscr(R_1, R')$.

4.2 Basic Definitions and Results

Following the ET framework, a declarative description in $Dscr(R_1, R_1)$ will be used as a definition part, while that in $Dscr(R_1, R_2)$ a query part. Given a definition part D and a query part Q, a transformation step rewriting Q into Q' is considered to be correct if and only if $D \cup Q$ and $D \cup Q'$ have the same meaning. By exploiting the fact that not a definite clause from R_1 to R_1 but only a definite clause from R_1 to R_2 is transformed, this subsection lays a simple yet general basis that not only enables precise discussion of the meanings of declarative descriptions, but also simplifies the verification of the correctness of expanding transformation.

In the sequel, given a set A, let $FP(A)$ denote the set of all finite subsets of A.

Definition 1 Given $U \subseteq \mathcal{G}_\Delta \times FP(\mathcal{G}_\Delta)$, the meaning of U, denoted by $M(U)$, is defined by

$$M(U) = \bigcup_{n=1}^{\infty} [T_U]^n(\emptyset),$$

where for any set $X \subseteq \mathcal{G}_\Delta$, $T_U(X)$ is the set

$$\{head \mid ((head, body) \in U) \& (body \subseteq X)\},$$

and for each $n \geq 2$, $[T_U]^n(\emptyset) = T_U([T_U]^{n-1}(\emptyset))$, and $[T_U]^1(\emptyset) = T_U(\emptyset)$. ■

Theorem 1 *Let* $g \in \mathcal{G}_\Delta$ *and* $U \subseteq \mathcal{G}_\Delta \times FP(\mathcal{G}_\Delta)$. *Then,* $g \in M(U)$ *if and only if there exists* $G \in FP(\mathcal{G}_\Delta)$ *such that* $(g, G) \in U$ *and* $G \subseteq M(U)$.

Proof

$$\begin{aligned}
& g \in M(U) \\
\iff & (\exists n \geq 1) : g \in [T_U]^n(\emptyset) \\
\iff & (\exists n \geq 1)(\exists G \in FP(\mathcal{G}_\Delta)) : \\
& \quad [((g, G) \in U) \ \& \ (G \subseteq [T_U]^{n-1}(\emptyset))] \\
\iff & (\exists G \in FP(\mathcal{G}_\Delta)) : \\
& \quad [((g, G) \in U) \ \& \ (G \subseteq M(U))].
\end{aligned}$$ ■

In the sequel, let $\{\mathcal{G}_1, \mathcal{G}_2\}$ be a partition of $\mathcal{G}_\Delta$ (i.e., $\mathcal{G}_1 \cup \mathcal{G}_2 = \mathcal{G}_\Delta$ and $\mathcal{G}_1 \cap \mathcal{G}_2 = \emptyset$); in addition, let $U_1 \subseteq \mathcal{G}_1 \times FP(\mathcal{G}_1)$ and $U_2 \subseteq \mathcal{G}_2 \times FP(\mathcal{G}_1)$.

Proposition 1 $M(U_1) = M(U_1 \cup U_2) \cap \mathcal{G}_1$.

Proof It will be shown by induction on n that $[T_{U_1}]^n(\emptyset) = [T_{(U_1 \cup U_2)}]^n(\emptyset) \cap \mathcal{G}_1$, for each $n \geq 1$.

Base case:

$$\begin{aligned}
& g \in [T_{U_1}]^1(\emptyset) \\
\iff & ((g, \emptyset) \in U_1) \\
\iff & ((g, \emptyset) \in (U_1 \cup U_2)) \ \& \ (g \in \mathcal{G}_1) \\
\iff & (g \in [T_{(U_1 \cup U_2)}]^1(\emptyset)) \ \& \ (g \in \mathcal{G}_1).
\end{aligned}$$

Induction Step:

$$\begin{aligned}
& g \in [T_{U_1}]^{n+1}(\emptyset) \\
\iff & (\exists G \in FP(\mathcal{G}_1)) : ((g, G) \in U_1) \\
& \quad \& \ (G \subseteq [T_{U_1}]^n(\emptyset)) \\
\iff & (\exists G \in FP(\mathcal{G}_1)) : ((g, G) \in U_1) \\
& \quad \& \ (G \subseteq ([T_{(U_1 \cup U_2)}]^n(\emptyset) \cap \mathcal{G}_1)) \\
& \quad \text{(by the induction hypothesis)} \\
\iff & (\exists G \in FP(\mathcal{G}_1)) : ((g, G) \in (U_1 \cup U_2)) \\
& \quad \& \ (g \in \mathcal{G}_1) \ \& \ (G \subseteq [T_{(U_1 \cup U_2)}]^n(\emptyset)) \\
\iff & (\exists G \in FP(\mathcal{G}_1)) : (g \in [T_{(U_1 \cup U_2)}]^{n+1}(\emptyset)) \\
& \quad \& \ (g \in \mathcal{G}_1).
\end{aligned}$$

As a result:

$$\begin{aligned}
& M(U_1 \cup U_2) \cap \mathcal{G}_1 \\
& = (\bigcup_{n=1}^{\infty} [T_{(U_1 \cup U_2)}]^n(\emptyset)) \cap \mathcal{G}_1 \\
& = \bigcup_{n=1}^{\infty} ([T_{(U_1 \cup U_2)}]^n(\emptyset) \cap \mathcal{G}_1) \\
& = \bigcup_{n=1}^{\infty} [T_{U_1}]^n(\emptyset) \\
& = M(U_1).
\end{aligned}$$ ■

Definition 2 The set $T(U_1, U_2)$ is defined by

$$T(U_1, U_2) = \{head \mid ((head, body) \in U_2) \ \& \ (body \subseteq M(U_1))\}.$$ ■

Proposition 2
$M(U_1 \cup U_2) \subseteq M(U_1) \cup T(U_1, U_2)$.

Proof Let $g \in M(U_1 \cup U_2)$. Then, by Theorem 1, there exists $G \in FP(\mathcal{G}_1)$ such that $(g, G) \in (U_1 \cup U_2)$ and $G \subseteq M(U_1 \cup U_2)$. Since $G \subseteq \mathcal{G}_1$, it follows from Proposition 1 that $G \subseteq M(U_1)$. Now suppose that $(g, G) \in U_1$. Then, by Theorem 1, $g \in M(U_1)$. Next, suppose that $(g, G) \in U_2$. It follows directly that $g \in T(U_1, U_2)$. ■

Proposition 3
$M(U_1 \cup U_2) \supseteq M(U_1) \cup T(U_1, U_2)$.

Proof Let $g \in M(U_1) \cup T(U_1, U_2)$. Suppose first that $g \in M(U_1)$. It follows from Theorem 1 that there exists $G \in FP(\mathcal{G}_1)$ such that $(g, G) \in U_1$ and $G \subseteq M(U_1)$. By Proposition 1, $M(U_1) \subseteq M(U_1 \cup U_2)$. So $G \subseteq M(U_1 \cup U_2)$, and, hence, by Theorem 1, $g \in M(U_1 \cup U_2)$.

Next suppose that $g \in T(U_1, U_2)$. Then there exists $G' \in FP(\mathcal{G}_1)$ such that $(g, G') \in U_2$ and $G' \subseteq M(U_1)$. As $M(U_1) \subseteq M(U_1 \cup U_2)$ (by Proposition 1), $G' \subseteq M(U_1 \cup U_2)$. Thus $g \in M(U_1 \cup U_2)$ by Theorem 1. ■

Theorem 2 $M(U_1 \cup U_2) = M(U_1) \cup T(U_1, U_2)$.

Proof The result follows from Propositions 2 and 3. ■

4.3 The Meanings of Declarative Descriptions

Let $P \in Dscr(R_1, R)$. Let $Pair(P)$ be the set

$$\{(Head(C\theta), Body(C\theta)) \mid (C \in P)\ \&\ (\theta \in \mathcal{S}_\Delta)\ \&\ (Head(C\theta) \in \mathcal{G}_\Delta)\ \&\ (Body(C\theta) \subseteq \mathcal{G}_\Delta)\}.$$

The meaning of P will now be defined.

Definition 3 The meaning $\mathcal{M}(P)$ of P is defined by $\mathcal{M}(P) = M(Pair(P))$. ■

Together with the results of the preceding subsection, the next definition and proposition will be used for proving the results of Subsection 5.2.

Definition 4 Let $D \in Dscr(R_1, R_1)$ and $Q \in Dscr(R_1, R_2)$. The set $\mathcal{T}(D, Q)$ is defined by

$$\mathcal{T}(D, Q) = T(Pair(D), Pair(Q)).$$ ■

Proposition 4 *Let $D \in Dscr(R_1, R_1)$ and Q, $Q_1, Q_2 \in Dscr(R_1, R_2)$. Then, if $\mathcal{T}(D, Q_1) = \mathcal{T}(D, Q_2)$, then $\mathcal{M}(D \cup Q \cup Q_1) = \mathcal{M}(D \cup Q \cup Q_2)$.*

Proof

$$\begin{aligned}
&\mathcal{T}(D, Q_1) = \mathcal{T}(D, Q_2) \\
&\iff T(Pair(D), Pair(Q_1)) \\
&\quad\ = T(Pair(D), Pair(Q_2)) \\
&\Longrightarrow T(Pair(D), Pair(Q_1)) \\
&\quad\ \cup T(Pair(D), Pair(Q)) \\
&\quad\ = T(Pair(D), Pair(Q_2)) \\
&\quad\ \cup T(Pair(D), Pair(Q)) \\
&\iff T(Pair(D), (Pair(Q_1) \cup Pair(Q))) \\
&\quad\ = T(Pair(D), (Pair(Q_2) \cup Pair(Q))) \\
&\iff T(Pair(D), Pair(Q \cup Q_1)) \\
&\quad\ = T(Pair(D), Pair(Q \cup Q_2)) \\
&\Longrightarrow T(Pair(D), Pair(Q \cup Q_1)) \\
&\quad\ \cup M(Pair(D)) \\
&\quad\ = T(Pair(D), Pair(Q \cup Q_2)) \\
&\quad\ \cup M(Pair(D)) \\
&\iff M(Pair(D) \cup Pair(Q \cup Q_1)) \\
&\quad\ = M(Pair(D) \cup Pair(Q \cup Q_2)) \\
&\quad\ \text{(by Theorem 2)} \\
&\iff M(Pair(D \cup Q \cup Q_1)) \\
&\quad\ = M(Pair(D \cup Q \cup Q_2)) \\
&\iff \mathcal{M}(D \cup Q \cup Q_1) = \mathcal{M}(D \cup Q \cup Q_2)).
\end{aligned}$$ ■

5 Expanding Transformation and Its Correctness

This section formally defines expanding transformation and proves the correctness thereof.

5.1 Expanding Transformation

In the rest of this paper, let $D \in Dscr(R_1, R_1)$ and assume that D contains the unit clause $eq(X, X) \leftarrow$ and does not contain any other clause from R_1 to $\{eq\}$; furthermore, let p be an n-ary predicate in R_1 and assume that

$$\begin{aligned}
C_{p_1}\colon&\quad p(s_1^1, \ldots, s_n^1) \leftarrow Bs_{p_1} \\
C_{p_2}\colon&\quad p(s_1^2, \ldots, s_n^2) \leftarrow Bs_{p_2} \\
&\quad \ldots \\
C_{p_m}\colon&\quad p(s_1^m, \ldots, s_n^m) \leftarrow Bs_{p_m}
\end{aligned}$$

be all the definite clauses from R_1 to $\{p\}$ in D.

Definition 5 (Expanding Transformation) Let C be a definite clause $H \leftarrow \{p(t_1, \ldots, t_n)\} \cup Bs$ from R_1 to R_2. For each i $(1 \le i \le m)$, let $\rho_i \in \mathcal{S}_\Delta$ be a variable-renaming substitution such that C and $C_{p_i}\rho_i$ do not have variables in common. Then, C can be transformed by *expanding* the body atom $p(t_1, \ldots, t_n)$ using D into m definite clauses $C_1', \ldots, C_m'$ from R_1 to R_2, where for each j $(1 \le j \le m)$, C_j' is the clause

$$H \leftarrow \{eq(t_1, s_1^j\rho_j), \ldots, eq(t_n, s_n^j\rho_j)\} \cup Bs_{p_j}\rho_j \cup Bs.$$

The set $\{C_1', \ldots, C_m'\}$ will be denoted by

$$Expand(C, p(t_1, \ldots, t_n), D, \langle (C_{p_1}, \rho_1), (C_{p_2}, \rho_2), \ldots, (C_{p_m}, \rho_m) \rangle),$$

and will be called a *result of transforming C by expanding $p(t_1, \ldots, t_n)$ using D.* ■

5.2 Correctness of Expanding Transformation

In the sequel, assume that C is a definite clause

$$H \leftarrow \{p(t_1, \ldots, t_n)\} \cup Bs$$

from R_1 to R_2; $Sel(C)$ denotes the body atom $p(t_1,\ldots,t_n)$ of C; $\rho_1,\rho_2,\ldots,\rho_m$ are variable-renaming substitutions in $\mathcal{S}_\Delta$ such that for each i $(1 \le i \le m)$, C and $C_{p_i}\rho_i$ have no variable in common; and

$$Expand(C, p(t_1,\ldots,t_n), D, \langle (C_{p_1},\rho_1),(C_{p_2},\rho_2),\ldots,(C_{p_m},\rho_m)\rangle) = \{C'_1,\ldots,C'_m\}.$$

Proposition 5

$$\mathcal{T}(D,\{C\}) \subseteq \mathcal{T}(D,\{C'_1,\ldots,C'_m\}).$$

Proof Let $g \in \mathcal{T}(D,\{C\})$. Then, there exists $\theta \in \mathcal{S}_\Delta$ such that $g = H\theta$ and $(\{Sel(C)\theta\} \cup Bs\theta) \subseteq M(Pair(D))$. Since $Sel(C)$ belongs to $M(Pair(D))$, it follows directly from Theorem 1 that there exists $G \in FP(\mathcal{G}_1)$ such that $(Sel(C)\theta, G) \in Pair(D)$ and $G \subseteq M(Pair(D))$. So there exist $\theta' \in \mathcal{S}_\Delta$ and i $(1 \le i \le m)$ such that $Sel(C)\theta = Head(C_{p_i})\theta'$ and $G = Bs_{p_i}\theta' \subseteq M(Pair(D))$. Now let $\rho_i^{-1} = \{x/y \mid y/x \in \rho_i\}$ and

$$\Theta = \{x/y \in \theta \mid x \text{ occurs in } C\} \cup \{x'/y' \in \rho_i^{-1}\theta' \mid x' \text{ occurs in } C_{p_i}\rho_i\}.$$

Since ρ_i is a variable-renaming substitution such that C and $C_{p_i}\rho_i$ have no variable in common, Θ is a well-defined substitution. Then, $Sel(C)\Theta = Sel(C)\theta = Head(C_{p_i})\theta' = Head(C_{p_i}\rho_i)\Theta$, $H\theta = H\Theta$, $Bs\theta = Bs\Theta$, and $Bs_{p_i}\theta' = (Bs_{p_i}\rho_i)\Theta$. Since $Sel(C)\Theta = Head(C_{p_i}\rho_i)\Theta$, it follows that for each j $(1 \le j \le n)$, $t_j\Theta = (s_j^i\rho_i)\Theta$, whence $eq(t_j, s_j^i\rho_i)\Theta \in M(Pair(D))$. Moreover, since $Bs\theta \subseteq M(Pair(D))$ and $Bs_{p_i}\theta' \subseteq M(Pair(D))$, $(Bs\Theta \cup (Bs_{p_i}\rho_i)\Theta) \subseteq M(Pair(D))$. Therefore $Head(C'_i)\Theta = H\Theta = g \in \mathcal{T}(D,\{C'_1,\ldots,C'_m\})$. ■

Proposition 6

$$\mathcal{T}(D,\{C\}) \supseteq \mathcal{T}(D,\{C'_1,\ldots,C'_m\}).$$

Proof Let $g \in \mathcal{T}(D,\{C'_1,\ldots,C'_m\})$. So there exist $\theta \in \mathcal{S}_\Delta$ and i $(1 \le i \le m)$ such that $g = H\theta$, $((Bs_{p_i}\rho_i)\theta \cup Bs\theta) \subseteq M(Pair(D))$, and for each j $(1 \le j \le n)$, $eq(t_j, s_j^i\rho_i)\theta \in M(Pair(D))$. Since $Bs_{p_i}(\rho_i\theta) = (Bs_{p_i}\rho_i)\theta \subseteq M(Pair(D))$ and $(Head(C_{p_i})(\rho_i\theta), Bs_{p_i}(\rho_i\theta)) \in Pair(D)$, $Head(C_{p_i})(\rho_i\theta) \in M(Pair(D))$ by Theorem 1. Since $eq(t_j, s_j^i\rho_i)\theta \in M(Pair(D))$, $t_j\theta = (s_j^i\rho_i)\theta$ for each j $(1 \le j \le n)$. Consequently, $Sel(C)\theta = p(t_1\theta,\ldots,t_n\theta) = p(s_1^i(\rho_i\theta), \ldots, s_n^i(\rho_i\theta)) = Head(C_{p_i})(\rho_i\theta) \in M(Pair(D))$. As $Bs\theta \subseteq M(Pair(D))$, it follows directly that $Head(C)\theta = H\theta = g \in \mathcal{T}(D,\{C\})$. ■

Proposition 7

$$\mathcal{T}(D,\{C\}) = \mathcal{T}(D,\{C'_1,\ldots,C'_m\}).$$

Proof The result follows from Propositions 5 and 6. ■

The main result of this Subsection is:

Theorem 3 (Correctness of Expanding Transformation) *Let $D \in Dscr(R_1,R_1)$ such that D contains the unit clause $eq(X,X) \leftarrow$ and does not contain any other clause from R_1 to $\{eq\}$. Let $Q \in Dscr(R_1,R_2)$. Let C be a definite clause from R_1 to R_2, and $Sel(C) \in Body(C)$. Let $\{C'_1,\ldots,C'_m\}$ be a result of transforming C by expanding $Sel(C)$ using D. Then $\mathcal{M}(D \cup Q \cup \{C\}) = \mathcal{M}(D \cup Q \cup \{C'_1,\ldots,C'_m\})$.*

Proof The result follows directly from Propositions 7 and 4. ■

6 Rewriting Rules and Their Correctness

The notion of meta-variable instantiation, based on which the applicability of a rewriting rule is determined, will be formulated. It is followed by the formal definition of a rewriting rule, its application, and its correctness.

Meta-Variable Instantiations A *meta-variable instantiation* is a mapping θ from V_2 to $\mathcal{T}_\Delta$ that satisfies the following three conditions:

(MVI-1) For each #-variable v, $\theta(v)$ is a variable.

(MVI-2) For any distinct #-variables v and v', $\theta(v) \neq \theta(v')$.

(MVI-3) For any &-variable u and #-variable v, $\theta(v)$ does not occur in $\theta(u)$.

Let $\hat{E}$ be an expression containing meta-variables ($\hat{E}$ can be, for example, a meta-term, a meta-atom, or a set of meta-atoms). Then, given a meta-variable instantiation θ, let $\hat{E}\theta$ denote the expression obtained from $\hat{E}$ by simultaneously replacing each occurrence of each meta-variable u in $\hat{E}$ with $\theta(u)$.

Rewriting Rules and Their Application A *rewriting rule r on R_1* takes the form

$$\hat{H} \rightarrow \hat{Bs}_1; \quad \cdots \quad \rightarrow \hat{Bs}_n,$$

where $n \ge 0$, and $\hat{H} \in \hat{\mathcal{A}}_1$ and the $\hat{Bs}_i \subseteq \hat{\mathcal{A}}_1$. For the sake of simplicity, the braces enclosing the meta-atoms in the right-hand side of a rewriting rule may be omitted; e.g., a rewriting

rule $\hat{H} \rightarrow \{\hat{B}_1, \ldots, \hat{B}_l\}$ will also be written as $\hat{H} \rightarrow \hat{B}_1, \ldots, \hat{B}_l$.

Let C be a definite clause $A \leftarrow \{B\} \cup Bs$ from R_1 to R_2. The rewriting rule r is said to be *applicable* to C at B by using a meta-variable instantiation θ, if and only if the following conditions are both satisfied:

(RRA-1) $\hat{H}\theta = B$.

(RRA-2) For any #-variable v, $\theta(v)$ occurs in neither A nor Bs.

When r is applied to C at B by using the meta-variable instantiation θ, it rewrites C into n definite clauses $C_1, \ldots, C_n$, where for each i $(1 \leq i \leq n)$, $C_i = (A \leftarrow \hat{Bs}_i\theta \cup Bs)$.

Correctness of Rewriting Rules Now what it means for a rewriting rule to be correct will be formally defined. Let $D \in Dscr(R_1, R_1)$. A rewriting rule r on R_1 is *correct with respect to D and R_2*, if and only if for any declarative description $Q \in Dscr(R_1, R_2)$ and any definite clauses $C, C_1, \ldots, C_n$ from R_1 to R_2, if r rewrites C into $C_1, \ldots, C_n$, then $\mathcal{M}(D \cup Q \cup \{C\}) = \mathcal{M}(D \cup Q \cup \{C_1, \ldots, C_n\})$.

7 General Rewriting Rules and Their Correctness

A class of rewriting rules, called *general rewriting rules*, with the widest applicability—the most general pattern of terms is used as the pattern of each predicate argument in their left-hand sides—will be introduced. Then their correctness will be proved based on Theorem 3.

7.1 General Rewriting Rules

Let ϱ be an injection from V_1 to V_2 such that for each $v \in V_1$, $\varrho(v)$ is a #-variable. In the sequel, for simplicity, assume that for each $v \in V_1$, $\varrho(v)$ has the same name as v except that $\varrho(v)$ begins with #; for instance, $\varrho(X) = \#X$ and $\varrho(Y) = \#Y$. Next, for any term t on Δ, let $t\varrho$ denote the meta-term on Δ obtained from t by simultaneously replacing each occurrence in t of each variable $u \in V_1$ with the #-variable $\varrho(u)$. Likewise, for any atom A on R, let $A\varrho$ denote the meta-atom on R obtained from A by simultaneously replacing each occurrence in A of each term t on Δ with $t\varrho$. Furthermore, for any set Bs of atoms on R, let $Bs\varrho = \{B\varrho \mid B \in Bs\}$.

In the sequel, refer to the declarative description D, the n-ary predicate p, and the definite clauses $C_{p_1}, \ldots, C_{p_m}$ of Section 5.

Definition 6 (General Rewriting Rule) The *general rewriting rule for p with respect to D*, denoted by $General(p, D)$, is defined as the rewriting rule

$$\begin{aligned} p(\&X_1, \ldots, \&X_n) &\rightarrow \hat{Bs}_{p_1}; \\ &\cdots \\ &\rightarrow \hat{Bs}_{p_m} \end{aligned}$$

on R_1, where $\&X_1, \ldots, \&X_n$ are arbitrary but distinct &-variables in V_2 and $\hat{Bs}_{p_i}$ is the set

$$\{eq(\&X_1, s_1^i\varrho), \ldots, eq(\&X_n, s_n^i\varrho)\} \cup Bs_{p_i}\varrho$$

for each i $(1 \leq i \leq m)$. ■

Referring to the declarative description D_{ap} of Section 2, for example, $General(app, D_{ap})$ is the rewriting rule

$$\begin{aligned} &app(\&X_1, \&X_2, \&X_3) \\ &\quad \rightarrow eq(\&X_1, [\,]), eq(\&X_2, \#Y), eq(\&X_3, \#Y); \\ &\quad \rightarrow eq(\&X_1, [\#A|\#X]), eq(\&X_2, \#Y), \\ &\qquad eq(\&X_3, [\#A|\#Z]), app(\#X, \#Y, \#Z), \end{aligned}$$

which is applicable at an *app*-atom of any form.

7.2 Correctness of General Rewriting Rules

The correctness of general rewriting rules will now be established.

Theorem 4 (Correctness of General Rewriting Rule) *The rewriting rule $General(p, D)$ is correct with respect to D and R_2.*

Proof Let $Q \in Dscr(R_1, R_2)$ and C be the definite clause $H \leftarrow \{p(t_1, \ldots, t_n)\} \cup Bs$ from R_1 to R_2. Suppose that $General(p, D)$ is applied to C at the body atom $p(t_1, \ldots, t_n)$ by using a meta-variable instantiation θ. Then $p(\&X_1, \ldots, \&X_n)\theta = p(t_1, \ldots, t_n)$, i.e., $\&X_i\theta = t_i$ for each i $(1 \leq i \leq n)$, and C is rewritten into m definite clauses $C_1, \ldots, C_m$ from R_1 to R_2, where for each j $(1 \leq j \leq m)$,

$$\begin{aligned} C_j &= (H \leftarrow \{(eq(\&X_1, s_1^j\varrho))\theta, \ldots \\ &\qquad \ldots, (eq(\&X_n, s_n^j\varrho))\theta\} \\ &\qquad \cup (Bs_{p_j}\varrho)\theta \cup Bs) \\ &= (H \leftarrow \{eq(\&X_1\theta, (s_1^j\varrho)\theta), \ldots \\ &\qquad \ldots, eq(\&X_n\theta, (s_n^j\varrho)\theta)\} \\ &\qquad \cup (Bs_{p_j}\varrho)\theta \cup Bs). \end{aligned}$$

Let $k \in \{1, \ldots, m\}$ and π_k be the substitution

$$\{v/\theta(\varrho(v)) \mid v \in V_1 \text{ and } v \text{ occurs in } C_{p_k}\}.$$

It is readily seen that

$$\begin{aligned} C_k = (H \leftarrow &\{eq(t_1, s_1^k\pi_k), \ldots, eq(t_n, s_n^k\pi_k)\} \\ &\cup Bs_{p_k}\pi_k \cup Bs). \end{aligned}$$

It will now be shown that π_k is a variable-renaming substitution such that C and $C_{p_k}\pi_k$ have no variable in common. Let $v \in V_1$. Since

$\varrho(v)$ is a #-variable, $v\pi_k = \theta(\varrho(v))$ is a variable that occurs neither in H nor Bs by Conditions (MVI-1) and (RRA-2). Moreover, by Condition (MVI-3) for θ, $v\pi_k$ does not occur in $\&X_l\theta = t_l$ for each l $(1 \leq l \leq n)$. So $v\pi_k$ does not occur in C; hence, C and $C_{p_k}\pi_k$ do not have any variable in common. Next let $u \in V_1$ such that $u \neq v$. Since ϱ is an injection from V_1 to V_2, $\varrho(u)$ and $\varrho(v)$ are different #-variables, whence $u\pi_k = \theta(\varrho(u)) \neq \theta(\varrho(v)) = v\pi_k$ by Condition (MVI-2). Consequently, π_k is a variable-renaming substitution. As a result,

$$Expand(C, p(t_1, \ldots, t_n), D, \langle (C_{p_1}, \pi_1), (C_{p_2}, \pi_2), \ldots, (C_{p_m}, \pi_m) \rangle) = \{C_1, \ldots, C_m\},$$

and it follows immediately from Theorem 3 that $\mathcal{M}(D \cup Q \cup \{C\}) = \mathcal{M}(D \cup Q \cup \{C_1, \ldots, C_m\})$. ■

8 Correctness of Expanding-based Rewriting Rules

Demonstrated in this section is the application of expanding transformation in justifying the correctness of an important class of rewriting rules, i.e., expanding-based rewriting rules, which subsumes the class of general rewriting rules discussed in Section 7.

8.1 Expanding-based Rewriting Rules

Rewriting rules whose correctness can be verified based solely on the correctness of expanding transformation and constraint solving for equality will be referred to as *expanding-based rewriting rules*. Every general rewriting rule is an expanding-based rewriting rule.

Given a definition part D' and a predicate p' such that the number of clauses in D' whose heads are a p'-atom is m', the rewriting rule $General(p', D')$ is always applicable at any p'-atom in the body of a clause and, when applied, always rewrites the clause into m' clauses. One can reduce the number of replacement clauses in a transformation step by constraining the applicability of a rewriting rule, i.e., by restricting the rule to be applicable only at atoms of some specific pattern, and specifying the application results only for atoms of this pattern. Minimizing the number of clauses resulting from a transformation step in general yields considerable improvement in computation efficiency [5].

As an illustration, consider the rewriting rule

r_2: $app(\&X, [\&E], [\&A, \&B|\&Z]) \rightarrow eq(\&X, [\&A|\#X]), app(\#X, [\&E], [\&B|\&Z])$.

This rule is only applicable at an ap-atom whose second and third arguments are a singleton list and a list with at least two elements, respectively; and, when applied to a clause, it transforms the clause into a single clause. As will be demonstrated in the next subsection, the correctness of this rule can be determined based solely on Theorem 3 and constraint solving for equality; it is therefore considered as an expanding-based rewriting rule. The rule r_1 of Subsection 2.2 is also an expanding-based rewriting rule.

8.2 Correctness of Expanding-based Rewriting Rules

To verify the correctness of the rule r_2 of the preceding subsection, consider a clause pattern

$\hat{C}_4$: $\hat{H} \leftarrow \{app(\&X, [\&E], [\&A, \&B|\&Z])\} \cup \hat{Bs}$,

where $\hat{H}$ represents an arbitrary atom; $\&A, \&B$, $\&E, \&X$ and $\&Z$ represent any arbitrary terms; and $\hat{Bs}$ represents an arbitrary set of atoms. Any clause of the form $\hat{C}_4$ can be expanded using D_{ap} into two corresponding clauses of the patterns

$\hat{C}_5$: $\hat{H} \leftarrow \{eq(\&X, [\,]), eq([\&E], \#Y), eq([\&A, \&B|\&Z], \#Y)\} \cup \hat{Bs}$

$\hat{C}_6$: $\hat{H} \leftarrow \{eq(\&X, [\#A|\#X]), eq([\&E], \#Y), eq([\&A, \&B|\&Z], [\#A|\#Z]), app(\#X, \#Y, \#Z)\} \cup \hat{Bs}$,

where $\#A, \#X, \#Y$ and $\#Z$ represent arbitrary but distinct variables that occur in none of the terms represented by $\&A, \&B, \&E, \&X$ and $\&Z$, and occur neither in the atom represented by $\hat{H}$ nor in any atom in the set represented by $\hat{Bs}$. By constraint solving for eq-atoms, any clause of the form $\hat{C}_5$ can be removed (any ground instantiation equalizing simultaneously the two arguments of each eq-atom in its body requires the variable represented by $\#Y$ to be instantiated into a singleton list and a list containing more than one element, which is a contradiction). Next, by examination of the eq-atoms in its body, $\hat{C}_6$ can be simplified into

$\hat{C}'_6$: $\hat{H} \leftarrow \{eq(\&X, [\&A|\#X]), app(\#X, [\&E], [\&B|\&Z])\} \cup \hat{Bs}$.

Now let r_2 be applied to a clause C from R_1 to R_2 and assume that this application transforms C into C'. Obviously, C must be a clause of the pattern $\hat{C}_4$ and, moreover, C' must be a corresponding clause of C of the pattern $\hat{C}'_6$. Next suppose that C_{Exp_1} and C_{Exp_2} are corresponding clauses of the patterns $\hat{C}_5$ and $\hat{C}_6$, respectively, of C. It is readily seen that the set $\{C_{Exp_1}, C_{Exp_2}\}$

is a result of expanding C using D_{ap}. Then it follows from Theorem 3 and the correctness of constraint solving for *eq*-atoms that for any $Q \in Dscr(R_1, R_2)$, $\mathcal{M}(D_{ap} \cup Q \cup \{C\}) = \mathcal{M}(D_{ap} \cup Q \cup \{C_{Exp_1}, C_{Exp_2}\}) = \mathcal{M}(D_{ap} \cup Q \cup \{C'\})$. Hence, the rule r_2 is correct with respect to D_{ap} and R_2.

9 Conclusions

The correctness of rewriting rules is a sufficient condition for the correctness of computation in the ET model. For practically checking their correctness, an appropriate foundation that facilitates the verification of systematic generation of rewriting rules [3] is necessary. The suitability of expanding transformation as such a foundation is explained, and the correctness of this operation is proved. Based on a framework for rigorously discussing the application and the correctness of rewriting rules, it is shown that the correctness of general rewriting rules—rewriting rules with the widest applicability—follows directly from the correctness of expanding transformation. The employment of expanding transformation in verifying the correctness of expanding-based rewriting rules—a larger and the most often-used class of rewriting rules—by manipulation of atom patterns and clause patterns is demonstrated.

Acknowledgement The first author was partially supported by the Thailand Research Fund (TRF). The second author was partly supported by Grant-in-Aid for Scientific Research (B)(2) #12480076.

References

[1] Akama, K., Kawaguchi, Y., and Miyamoto, E., *Equivalent Transformation for Equality Constraints on Multiset Domains* (in Japanese), J. Japanese Society for Artificial Intelligence, **13**(3), 395–403, 1998.

[2] Akama, K., Kawaguchi, Y., and Miyamoto, E., *Solving Logical Problems by Equivalent Transformation—Limitations of SLD Resolution* (in Japanese), J. Japanese Society for Artificial Intelligence, **13**(6), 936–943, 1998.

[3] Akama, K., Koike, H., and Miyamoto, E., *Program Synthesis from a Set of Definite Clauses and a Query*, Proc. 5th International Conference on Information Systems Analysis and Synthesis, Orlando, Florida, 1999.

[4] Akama, K., Nantajeewarawat, E., and Koike, H., *A Class of Rewriting Rules and Reverse Transformation for Rule-Based Equivalent Transformation*, Proc. 2nd International Workshop on Rule-Based Programming, Firenze, Italy, 2001.

[5] Akama, K., Shigeta, Y., and Miyamoto, E., *Solving Problems by Equivalent Transformation of Logic Programs*, Proc. 5th International Conference on Information Systems Analysis and Synthesis, Orlando, Florida, 1999.

[6] Akama, K., Shimizu, T., and Miyamoto, E., *Solving Problems by Equivalent Transformation of Declarative Programs* (in Japanese), J. Japanese Society for Artificial Intelligence, **13**(6), 944–952, 1998.

[7] Anutariya, C., Wuwongse, V., Nantajeewarawat, E., and Akama, K., *Towards Computation with RDF Elements*, Proc. International Symposium on Digital Libraries, Tsukuba, Japan, 1999.

[8] Burstall, R. M. and Darlington, J., *A Transformation System for Developing Recursive Programs*, J. ACM, **24**(1), 44–67, 1977.

[9] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1987.

[10] Nantajeewarawat, E., Wuwongse, V., Anutariya, C., Akama, K., and Thiemjarus, S., *Towards Reasoning with UML Diagrams Based-on Declarative Description Theory*, Proc. International Conference on Intelligence Technologies, Bangkok, Thailand, 2000.

[11] Pettorossi, K. and Proietti, M., *Transformation of Logic Programs*, Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5, Oxford University Press, pp. 697–787, 1998.

[12] Robinson, J. A., *A Machine-Oriented Logic Based on the Resolution Principle*, J. ACM, **12**, 23–41, 1965.

[13] Suita, K., Akama, K., and Miyamoto, E., *Solving Constraint Satisfaction Problems by Equivalent Transformation* (in Japanese), IEICE Tech. Report SS96-18, pp. 1–8, 1996.

[14] Suita, K., Akama, K., and Miyamoto, E., *Constructing Natural Language Understanding Systems Based-on Equivalent Transformation* (in Japanese), IEICE Tech. Report SS97-35, pp. 23–30, 1997.

[15] Tamaki, K. and Sato, T., *Unfold/Fold Transformation of Logic Programs*, Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, 1984.

[16] Wuwongse, V., Anutariya, C, Akama, K., and Nantajeewarawat, E., *XML Declarative Description: A Language for the Semantic Web*, IEEE Intelligent Systems, **16**(3), 54–65, 2001.

Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 16 pages

A Class of Rewriting Rules and Reverse Transformation for Rule-based Equivalent Transformation

Kiyoshi Akama [1,2]

Center for Information and Multimedia Studies
Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan

Ekawit Nantajeewarawat [3,4]

IT Program, Sirindhorn International Institute of Technology
Thammasat University, Rangsit Campus
P.O. Box 22, Thammasat-Rangsit Post Office, Pathumthani 12121, Thailand

Hidekatsu Koike [5]

Division of System and Information Engineering
Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan

Abstract

In the rule-based equivalent transformation (RBET) paradigm, where computation is based on meaning-preserving transformation of declarative descriptions, a set of rewriting rules is regarded as a program. The syntax for a large class of rewriting rules is determined. The incorporation of meta-variables of two different kinds enables precise control of rewriting-rule instantiations. As a result, the applicability of rewriting rules and the results of rule applications can be rigorously specified. A theoretical basis for justifying the correctness of rewriting rules is established. Reverse transformation operation in the RBET framework is discussed, and it is shown that a correct rewriting rule is reversible, i.e., a correct rewriting rule can in general be constructed by syntactically reversing another correct rewriting rule.

[1] Akama was partly supported by Grant-in-Aid for Scientific Research (B)(2) #12480076.
[2] Email: akama@cims.hokudai.ac.jp
[3] Nantajeewarawat was supported partially by the Thailand Research Fund.
[4] Email: ekawit@siit.tu.ac.th
[5] Email: koke@cims.hokudai.ac.jp

1 Introduction

Rule-based equivalent transformation of declarative descriptions (RBET) [1] is a new promising method of problem solving. In the RBET framework, a problem is formulated as a *declarative description*, represented by the union of two sets of definite clauses, one of which is called the *definition part*, and the other the *query part*. The definition part provides general knowledge about the problem domain and descriptions of some specific problem instances. The query part specifies a question regarding the content of the definition part. From the definition part, a set of *rewriting rules*—rules for transforming declarative descriptions—is prepared. The problem is then solved by transforming the query part successively, using the prepared rewriting rules, into another set of definite clauses from which the answers to the specified question can be obtained easily and directly.

Example 1.1 Consider a simple problem formulated as the union of a definition part D_{init} consisting of the four definite clauses

$$initial(X, Z) \leftarrow append(X, Y, Z)$$
$$append([\,], Y, Y) \leftarrow$$
$$append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)$$
$$equal(X, X) \leftarrow$$

and a query part Q containing only the definite clause

$$C_1\colon \quad ans(X) \leftarrow initial(X, [1, 2, 3]), initial(X, [1, 3, 5]).$$

To solve this problem, i.e., to find the answers to the query part Q, by means of RBET, Q will be transformed successively, using some rewriting rules prepared from D_{init}, until the simpler query part Q' consisting of the two unit clauses

$$ans([\,]) \leftarrow$$
$$ans([1]) \leftarrow$$

is obtained, from which the answers, i.e., $X = [\,]$ and $X = [1]$, can be directly drawn. One possible successive transformation of Q into Q' is demonstrated in the appendix. □

A rewriting rule specifies, in its left-hand side, a pattern of atomic formulas (atoms) to which it can be applied, and defines the result of its application by specifying, in its right-hand side, one or more patterns of replacement atoms. The rule is applicable to a definite clause when the pattern in the left-hand side matches atoms contained in the body of the clause—in other words, when atoms contained in the body of the clause are instances of the specified pattern. When applied, the rule rewrites the clause into a number of clauses, resulting from replacing the matched body atoms with instances of the patterns in the right-hand side of the rule. Determination of rule applicability by pattern matching, rather than unification, allows one to tailor a rewriting rule for some specific pattern of atoms for the sake of computation efficiency.

Illustrations of rewriting rules are deferred until Section 2.

The crucial roles of atom patterns in determining rule applicability and specifying the results of rule applications necessitate an appropriate syntactic structure for representing the patterns in such a way that their instantiations can be precisely and suitably controlled. For this purpose, the notion of *meta-atom* is introduced. Meta-atoms have the same structure as usual atoms except that two kinds of *meta-variables*—&-variables and #-variables—are used instead of ordinary variables. The two kinds of meta-variables have different instantiation characteristics. Not only do the differences allow precise specifications of rewriting rules; they enable rigorous investigation of several important properties of several kinds of transformation steps, e.g., correctness of expanding transformation [7], and, moreover, as shown in [3], systematic generation of correct rewriting rules from a problem specification.

In the RBET framework, the correctness of computation relies solely on the correctness of each transformation step. Given a declarative description $D \cup Q$, where D and Q represent the definition part and the query part, respectively, of a problem, the query part Q is said to be transformed correctly in one step into a new query part Q' by an application of a rewriting rule, if and only if the declarative descriptions $D \cup Q$ and $D \cup Q'$ are equivalent, i.e., they have the same declarative meaning. A rewriting rule is considered to be correct, if and only if its application always results in a correct transformation step. A correct rewriting rule will be referred to as an *Equivalent Transformation rule (ET rule)*. If ET rules are employed in all transformation steps, the answers obtained by means of RBET are guaranteed to be correct.

1.1 Comparison Between RBET and the Logic Programming Paradigm

Computation

Although declarative descriptions considered in this paper have the same form as definite logic programs [5], computation in RBET differs significantly from that in logic programming. Computation in logic programming is based on logical deduction—computation is viewed as the process of constructing, based on the resolution principle [10], a proof of an existentially quantified query by finding variable substitutions, called *computed substitutions*, that make the query follow logically from a given logic program. In RBET, by contrast, computation is regarded as transformation of declarative descriptions rather than logical deduction.

Separation of Programs from Declarative Descriptions

In logic programming, a set of definite clauses has a dual function: it serves as a declarative description of a problem—it declaratively represents the knowledge about the problem domain and defines what the problem is—while at the same time functions as a program—it specifies how to solve the problem. The programming character of a set of definite clauses arises from viewing

it as a description of a search whose structure is determined by interpreting the logical connectives and quantifiers as fixed search instructions [6]. The procedural expressive power of a logic programming language, such as Prolog, is limited by such fixed procedural interpretation and the fixed search strategy embedded in the proof procedure associated with the language.

In the RBET framework, instead of a set of definite clauses, a set of rewriting rules is regarded as a program. The procedural interpretation of definite clauses can be realized using rewriting rules of a basic kind, called *unfolding-based rewriting rules* [1]. However, several other rewriting rules can additionally be employed in RBET, thereby a wider variety of computation paths are allowed and a more efficient program can consequently be achieved [1]. The use of a set of rewriting rules as a program also enables flexible computation—an effective control strategy can be materialized by means of, for example, rule-firing control and user-defined priority-based selection of rules [1].

Theoretical Foundation for Correctness

While the correctness of computation in RBET is based solely on meaning preservation of declarative descriptions, the correctness of computation in logic programming is grounded upon the logical consequence relation ($\models$), i.e., given a logic program P and an atom q, a computed substitution θ is correct if and only if $P \models \forall(q\theta)$. The notion of logical consequence in turn relies on the elementary concepts, e.g., the concepts of interpretation, satisfaction, and model, of the model theory associated with first-order logic. These concepts are not necessary in the RBET framework.

The correctness of computation in logic programming cannot be guaranteed by the correctness of inference rules solely; it also depends on the computation procedure employed. When the computation procedure is improved or extended, the correctness of the procedure as a whole has to be proven. In comparison, to verify the correctness of computation in RBET, it suffices to prove the correctness of each individual rewriting rule. A program in the RBET framework can therefore be decomposed; consequently, RBET-based systems are amenable to modification and extension.

1.2 Comparison Between RBET and Program Transformation in Logic Programming

Objectives and Transformed Parts

The objective of RBET is different from that of program transformation in logic programming (PT) [8,9]. While RBET is a method for computing the answers to a question with respect to a given definition part, PT is a methodology for deriving an efficient logic program from the definition part. Let a definition part D_0 be given. In RBET, to compute the answers to a query part Q_0 with respect to D_0, one constructs from Q_0, by successive application of rewriting rules prepared from D_0, a sequence $Q_0, \ldots, Q_n$ such that for each

i $(0 \leq i < n)$, $D_0 \cup Q_i$ and $D_0 \cup Q_{i+1}$ have the same declarative meaning and the answers can be directly obtained from Q_n. The definition part D_0 is unchanged throughout the transformation process. In comparison, in PT only the definition part is transformed. That is, from D_0, which is regarded as the initial logic program, one constructs by using transformation rules, such as the unfolding and folding rules, a sequence of logic programs $D_0, \ldots, D_m$ such that D_0 and D_m yield the same answers to some class of queries, but D_m is more efficient than D_0; then, when a query in that class is given, the program D_m will be used for computing the answers to the query by means of some proof procedure.

Example 1.2 Consider the definition part D_{init} of Example 1.1. Following PT, D_{init} may be transformed successively, using the unfolding and folding rules, into the logic program D'_{init}:

$$initial([\,], Y) \leftarrow$$
$$initial([A|X], [A|Z]) \leftarrow initial(X, Z)$$
$$append([\,], Y, Y) \leftarrow$$
$$append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)$$

D_{init} and D'_{init} have the same declarative meaning with respect to the predicates *initial* and *append*; however, computing the answers to a query containing the predicate *initial* using D'_{init} requires fewer number of resolution steps than using D_{init}. □

In PT the efficiency of the program resulting from a transformation process, rather than the transformation process itself, is the primary concern. In RBET, on the other hand, as transformation is the main computation mechanism, transformation processes are required to be efficient. The efficiency of a transformation process in RBET is achieved by the employment of efficient rewriting rules and appropriate rule-application control strategies [1].

Correctness and Independence of Rules

In PT, a transformation step which derives D_{k+1} from a transformation sequence $D_0, \ldots, D_k$ is correct, if and only if for each query q containing only predicate symbols which occur in D_k, D_k and D_{k+1} provide the same answers to q. The correctness of a transformation step in PT can in general not be determined independently; e.g., the correctness of a folding step deriving D_{k+1} from a transformation sequence $D_0, \ldots, D_k$ requires some conditions to ensure that enough unfolding steps have been performed in the sequence $D_0, \ldots, D_k$ [9]. The next example shows that an application of the folding rule may yield an incorrect transformation step.

Example 1.3 Refer to the definition part D_{init} of Example 1.1. Folding the first clause, i.e.,

$$initial(X, Z) \leftarrow append(X, Y, Z),$$

using itself results in the logic program D''_{init}:

$$initial(X, Z) \leftarrow initial(X, Z)$$
$$append([\,], Y, Y) \leftarrow$$
$$append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)$$

Since the meaning of the predicate *initial* defined in D_{init} is lost in D''_{init}, this transformation step does not preserve the answers to queries concerning the predicate *initial* and is therefore not correct. □

In RBET, by contrast, since only a query part, which depends exclusively on a fixed definition part, is transformed, the correctness of a transformation step can be justified independently, i.e., given a definition part D, the correctness of a transformation step deriving a query part Q_{j+1} from a query part Q_j is determined by the meanings of $D \cup Q_j$ and $D \cup Q_{j+1}$ solely, regardless of its preceding transformation steps. Consequently, the correctness of a rewriting rule can also be determined independently in the RBET framework. Such independence of rewriting rules is apparently desirable for the construction of large-scale rule-based systems.

1.3 Objectives of the Paper

Syntax for Rewriting Rules. The first objective of this paper is to determine appropriate syntax for a large class of rewriting rules. The syntactic structure of rewriting-rule components as well as their instantiations should be suitably defined in order that they can be used to precisely specify rule applicability and the results of rule applications.

Theoretical Framework for Correctness of Rewriting Rules. The next objective is to establish, based on meaning-preserving transformation of declarative descriptions rather than logical inference, a theoretical framework for discussing the correctness of rewriting rules.

Reverse Transformation. The third objective is to introduce the reverse transformation operation, and to show that in the RBET framework an ET rule is reversible, i.e., one can obtain a rewriting rule the operation of which reverses that of another rewriting rule by syntactically reversing the latter rewriting rule, and the correctness of the former depends solely on the correctness of the latter.

Section 2 explains the necessity of meta-variables, and provides introductory examples of rewriting rules, reverse transformation, and reverse rewriting rules. Section 3 defines preliminary syntactic components, which are used for defining declarative descriptions and their meanings in Section 4, and rewriting rules, their applications, and their correctness in Section 5. Section 6 investigates the correctness of reverse rewriting rules.

2 Meta-Variables and Reverse Rewriting Rules

The need for the use of meta-variables of two distinct kinds for specifying patterns of atoms, and the necessity of conditions for regulating meta-variable instantiations will be described first. Reverse transformation operation and reverse rewriting rules will then be introduced.

2.1 Need for Meta-Variables of Two Kinds

Consider the definition part D_{init} and the query parts Q and Q' of Example 1.1. As the first step of a possible transformation sequence leading to Q', the clause

$$C_1\colon \quad ans(X) \leftarrow initial(X, [1,2,3]), initial(X, [1,3,5])$$

in Q may be transformed by replacing its first body atom with $append(X, Y, [1, 2, 3])$, resulting in the clause

$$C_2\colon \quad ans(X) \leftarrow append(X, Y, [1,2,3]), initial(X, [1,3,5]).$$

This transformation step is correct since $D_{init} \cup \{C_1\}$ and $D_{init} \cup \{C_2\}$ have the same meaning.

The above transformation step can be described by the rewriting rule

$$r_1\colon \quad initial(\&X, \&Z) \;\rightarrow\; append(\&X, \&Y, \&Z),$$

where the arrow "$\rightarrow$" intuitively means "can be replaced with" and the left-hand side and the right-hand side of r_1 specify the pattern of atoms to which the rule is applicable and the pattern of replacement atoms, respectively. The symbols $\&X, \&Y$ and $\&Z$ are used in r_1 as instantiation wild cards, i.e., each of them can be instantiated into an arbitrary term, and also as equality constraints, i.e., each occurrence of the same wild card must be instantiated into the same term. By instantiating $\&X, \&Y$ and $\&Z$ into the terms X, Y and $[1, 2, 3]$, respectively, the pattern in the left-hand side matches the first body atom of C_1 and that in the right-hand side is instantiated into the first body atom of C_2—that is, by applying r_1 to the first body atom of C_1 using this instantiation, C_1 is transformed into C_2.

The dual role of the symbols $\&X, \&Y$ and $\&Z$ as wild cards and equality constraints is reminiscent of the concept of variable. Notwithstanding, these symbols should be distinguished from ordinary variables that are used in definite clauses since they are used differently; for example, they can be instantiated into ordinary variables but they are not substituted for ordinary variables in any substitution application. To emphasize the differences, the symbols $\&X, \&Y$ and $\&Z$ will be regarded as meta-variables, and will be referred to as &-variables.

However, the rewriting rule r_1 does not always specify a correct transformation step. For example, the application of r_1 to the first body atom of C_1 by instantiating $\&Y$ into the variable X transforms C_1 into the clause

$$C_3\colon \quad ans(X) \leftarrow append(X, X, [1,2,3]), initial(X, [1,3,5]),$$

but $D_{init} \cup \{C_1\}$ and $D_{init} \cup \{C_3\}$ have different meanings.

To ensure a correct transformation step, some restrictions on rule instantiations are required. Another kind of meta-variable, called #-variables, is introduced for this purpose. As an example, a #-variable, $\#Y$, will be used instead of the &-variable $\&Y$ in the right-hand side of r_1, i.e., the rule

$$r_2\colon \quad initial(\&X, \&Z) \ \rightarrow \ append(\&X, \#Y, \&Z)$$

will be used instead of r_1. Then, any instantiation of this rule is regulated in such a way that the #-variable $\#Y$ can only be instantiated into an ordinary variable that does not appear in the other part of the clause resulting from an application of the rule. This instantiation constraint precludes the instantiation of $\#Y$ into the ordinary variable X when the rule r_2 is applied to the first body atom of C_1; as a result, the transformation of C_1 into C_3 is prevented.

2.2 Reverse Transformation

In the RBET framework, the reverse of a correct transformation step is always a correct transformation step. For instance, from the step transforming C_1 into C_2 illustrated in the preceding subsection, one can have the reverse step transforming C_2 into C_1, which may be described by the rewriting rule

$$r_3\colon \quad append(\&X, \&Y, \&Z) \ \rightarrow \ initial(\&X, \&Z),$$

and the correctness of the latter step follows from the correctness of the former step. In general, however, the application of the rule r_3 may result in an incorrect transformation step. For example, by instantiating the &-variable $\&Y$ into X, the application of r_3 to the first body atom of the clause C_3 of the previous subsection yields an incorrect transformation step deriving C_1 from C_3.

Again the employment of meta-variables of the two kinds, with different instantiation characteristics, remedies this problem. Instead of using r_3, the transformation of C_2 into C_1 can be described using the rewriting rule

$$r_4\colon \quad append(\&X, \#Y, \&Z) \ \rightarrow \ initial(\&X, \&Z),$$

while the application of r_4 to the first body atom of C_3 can be ruled out by appropriately restricting the instantiation of the #-variable $\#Y$, i.e., $\#Y$ is only allowed to be instantiated into a variable that does not occur in the other part of C_3.

Rigorous description of rewriting rules and their applications demands precise conditions for instantiations of meta-variables in rule applications. For the sake of generality and regularity, the conditions should not be specialized for any particular case, but common to all rewriting rules. Such common conditions will be defined in Section 5 (Conditions (MVI-1), (MVI-2), (MVI-3) and (RRA-2)).

Notice that the rule r_4 can be obtained by simply reversing the rule r_2 of the preceding subsection. It will be shown in Section 6 that in the RBET

framework a correct rewriting rule can in general be constructed by reversing another correct rewriting rule.

3 Basic Syntactic Components

The alphabet used in the paper will now be given; then, the notions of term and atom, which are basic components of definite clauses and declarative descriptions, and those of meta-term and meta-atom, which are used for specifying patterns of terms and atoms, respectively, will be defined.

Alphabet

An *&-variable* is a variable that begins with the symbol &; for example, $\&N$ and $\&X$ are &-variables. A *#-variable* is a variable that begins with the symbol #; for example, $\#X$ and $\#Y$ are #-variables. An &-variable as well as a #-variable is called a *meta-variable*. An ordinary variable is assumed to begin with neither & nor #.

Throughout the paper, an alphabet $\Delta = \langle K, F, V, R \rangle$ is assumed, where K is a set of constants, including integers and *nil*; F a set of functions, including the binary function *cons*; V is the disjoint union of two sets

- V_1 of ordinary variables,
- V_2 of meta-variables;

and R is the union of two mutually disjoint sets of predicates

- $R_1 = \{initial, append, equal, \ \ldots\}$,
- $R_2 = \{ans, yes, \ \ldots\}$.

When no confusion is possible, an ordinary variable in V_1 and a meta-variable in V_2 will be simply called a variable and a meta-variable respectively.

Terms, Meta-Terms, Atoms, and Meta-Atoms

Usual first-order terms on $\langle K, F, V_1 \rangle$ and on $\langle K, F, V_2 \rangle$ will be referred to as *terms* and *meta-terms*, respectively, on Δ. Given $R' \subseteq R$, usual first-order atoms on $\langle K, F, V_1, R' \rangle$ and on $\langle K, F, V_2, R' \rangle$ will be referred to as *atoms on* R' and *meta-atoms on* R', respectively. For example, assume that $\{X, Y\} \subseteq V_1$ and $\{\&X, \#Y\} \subseteq V_2$. Then, *nil*, X and $cons(X, cons(Y, nil))$ are terms on Δ; *nil*, $\&X$ and $cons(\&X, cons(\#Y, nil))$ are meta-terms on Δ; $initial(X, cons(X, cons(Y, nil)))$ is an atom on R_1; and $initial(\&X, cons(\&X, cons(\#Y, nil)))$ is a meta-atom on R_1. The standard Prolog notation for lists is adopted; e.g., $[X, Y]$ and $[7, \#X|\&Y]$ are abbreviations for the term $cons(X, cons(Y, nil))$ and the meta-term $cons(7, cons(\#X, \&Y))$, respectively.

First-order atoms on $\langle K, F, \emptyset, R \rangle$ are called *ground atoms* on Δ. In the sequel, let $\mathcal{T}$ be the set of all terms on Δ, and $\mathcal{G}$ the set of all ground atoms on Δ; also let $\mathcal{A}_i$ and $\hat{\mathcal{A}}_i$ be the set of all atoms and the set of all meta-atoms, respectively, on R_i, where $i \in \{1, 2\}$.

4 Declarative Descriptions and Their Meanings

In general, the RBET framework can deal with several data structures other than usual first-order terms, e.g., multisets, strings; and a declarative description can be represented by a set of definite clauses extended with these data structures [2,4]. For simplicity, however, only usual terms are used in this paper; that is, a declarative description is a set of usual definite clauses. Definite clauses and declarative descriptions considered herein as well as the meanings of declarative descriptions will now be defined.

Definite Clauses and Declarative Descriptions

A *definite clause* C *on* Δ is an expression of the form $A \leftarrow Bs$, where A is an atom on R and Bs is a (possibly empty) set of atoms on R. The atom A is called the *head* of C, denoted by $head(C)$; the set Bs is called the *body* of C, denoted by $Body(C)$; each element of $Body(C)$ is called a *body atom* of C. When $Body(C) = \emptyset$, C will be called a *unit clause*. The set notation is used in the right-hand side of C so as to stress that the order of the atoms in $Body(C)$ is immaterial. However, for the sake of simplicity, the braces enclosing the body atoms in the right-hand side of a definite clause will often be omitted; e.g., the definite clause $ans(X) \leftarrow \{append(Y, X, Z), initial(Y, Z)\}$ will often be written as $ans(X) \leftarrow append(Y, X, Z), initial(Y, Z)$.

Let $i \in \{1, 2\}$. A definite clause C is said to be *from* R_1 *to* R_i, if and only if $Body(C) \subseteq \mathcal{A}_1$ and $head(C) \in \mathcal{A}_i$. A *declarative description from* R_1 *to* R_i is a set of definite clauses from R_1 to R_i. The set of all declarative descriptions from R_1 to R_i will be denoted by $Dscr(R_1, R_i)$.

Meanings of Declarative Descriptions

Let $\mathcal{S}$ be the set of all substitutions on $\langle K, F, V_1 \rangle$. The application of a substitution θ to an expression E (which can be, for example, a term, an atom, a set of atoms, or a definite clause) will be denoted by $E\theta$. Given a declarative description $P \in Dscr(R_1, R_i)$, the mapping T_P on $2^{\mathcal{G}}$ is given by

$$T_P(X) = \{head(C\theta) \mid (C \in P) \;\&\; (\theta \in \mathcal{S}) \;\&\; (head(C\theta) \in \mathcal{G}) \;\&\; (Body(C\theta) \subseteq X)\},$$

and then, the meaning of P, denoted by $\mathcal{M}(P)$, is defined by

$$\mathcal{M}(P) = T_P^1(\emptyset) \cup T_P^2(\emptyset) \cup T_P^3(\emptyset) \cup \cdots = \bigcup_{n=1}^{\infty} T_P^n(\emptyset),$$

where $T_P^1(\emptyset) = T_P(\emptyset)$ and $T_P^n(\emptyset) = T_P(T_P^{n-1}(\emptyset))$ for each $n \geq 2$.

5 Rewriting Rules, Their Applications, and Their Correctness

The syntax for a large class of rewriting rules is next presented. Coupled with some restrictions on meta-variable instantiations, this syntax enables one to

control the applicability of rewriting rules and to specify the results of rule applications in a precise way.

Syntax of Rewriting Rules
A *rewriting rule on* R_1 takes the form

$$\begin{aligned} \hat{H}s &\rightarrow \hat{B}s_1; \\ &\cdots \\ &\rightarrow \hat{B}s_n, \end{aligned}$$

where $n \geq 0$, and $\hat{H}s$ and the $\hat{B}s_i$ are subsets of $\hat{\mathcal{A}}_1$. For the sake of simplicity, the braces enclosing the meta-atoms in each side of a rewriting rule may be omitted; e.g., the rewriting rule $\{initial(\&X, \&Z)\} \rightarrow \{append(\&X, \#Y, \&Z)\}$ will also be written as $initial(\&X, \&Z) \rightarrow append(\&X, \#Y, \&Z)$.

Meta-Variable Instantiations
A *meta-variable instantiation* is a mapping θ from V_2 to $\mathcal{T}$ that satisfies the following three conditions:

(MVI-1) For each #-variable v, $\theta(v)$ is a variable.

(MVI-2) For any distinct #-variables v and v', $\theta(v) \neq \theta(v')$.

(MVI-3) For any &-variable u and #-variable v, $\theta(v)$ does not occur in $\theta(u)$.

Let $\hat{E}$ be an expression containing meta-variables ($\hat{E}$ can be, for example, a meta-term, a meta-atom, or a set of meta-atoms). Then, given a meta-variable instantiation θ, let $\hat{E}\theta$ denote the expression obtained from $\hat{E}$ by simultaneously replacing each occurrence of each meta-variable u in $\hat{E}$ with $\theta(u)$.

Applicability of Rewriting Rules
Let r be a rewriting rule on R_1

$$\begin{aligned} \hat{H}s &\rightarrow \hat{B}s_1; \\ &\cdots \\ &\rightarrow \hat{B}s_n, \end{aligned}$$

where $n \geq 0$, and $\hat{H}s$ and the $\hat{B}s_i$ are subsets of $\hat{\mathcal{A}}_1$. Let C be a definite clause

$$A \leftarrow Bs \cup Bs'$$

from R_1 to R_2. The rewriting rule r is said to be *applicable* to C at Bs by using a meta-variable instantiation θ, if and only if the following conditions are both satisfied:

(RRA-1) $\hat{H}s\theta = Bs$.

(RRA-2) For any #-variable v, $\theta(v)$ occurs in neither A nor Bs'.

When r is applied to C at Bs by using the meta-variable instantiation θ, it rewrites C into n definite clauses $C_1, \ldots, C_n$, where for each i $(1 \leq i \leq n)$,

$$C_i = (A \leftarrow \hat{Bs}_i\theta \cup Bs').$$

When Bs is a singleton set $\{B\}$, the application of r to C at Bs will also be referred to as the application of r to the body atom B of C.

When there are more than one applicable rewriting rule, one of them will be nondeterministically selected; hence, computation in RBET is nondeterministic.

Examples illustrating the application of rewriting rules are given below.

Example 5.1 Refer to the rewriting rules r_2 and r_4 and the definite clauses C_1, C_2 and C_3 of Section 2. Let $\theta : V_2 \rightarrow \mathcal{T}$ such that $\theta(\&X) = X$, $\theta(\#Y) = Y$, $\theta(\&Z) = [1,2,3]$ and θ satisfies Conditions (MVI-1), (MVI-2) and (MVI-3). Then, since Y occurs in neither the head nor the second body atom of C_1, r_2 can be applied to C_1 at $\{initial(X,[1,2,3])\}$ by using θ, and this application rewrites C_1 into C_2. Likewise, the application of r_4 to C_2 at $\{append(X,Y,[1,2,3])\}$ by using θ rewrites C_2 into C_1. Now consider the clause C_3. The rule r_4 is not applicable to C_3, since every $\sigma : V_2 \rightarrow \mathcal{T}$ such that

$$append(\&X, \#Y, \&Z)\sigma = append(X, X, [1,2,3])$$

requires that $\sigma(\&X) = X = \sigma(\#Y)$, violating Condition (MVI-3). □

Example 5.2 Consider the rewriting rule

$$\begin{aligned} r_5\colon\ & append(\&X, \&Y, \&Z) \\ & \rightarrow equal(\&X, [\,]), equal(\&Y, \&Z); \\ & \rightarrow equal(\&X, [\#A|\#X]), equal(\&Z, [\#A|\#Z]), \\ & \quad append(\#X, \&Y, \#Z), \end{aligned}$$

and the clause

$$C_4\colon\ ans(X) \leftarrow append(X, [E], [1,2]).$$

The application of the rule r_5 to C_4 transforms C_4 into the two definite clauses

$$C_5\colon\ ans(X) \leftarrow equal(X, [\,]), equal([E], [1,2])$$

$$\begin{aligned} C_6\colon\ & ans(X) \leftarrow equal(X, [A1|X1]), equal([1,2], [A1|Z1]), \\ & \quad append(X1, [E], Z1) \end{aligned}$$

by using a meta-variable instantiation θ such that $\theta(\&X) = X$, $\theta(\&Y) = [E]$, $\theta(\&Z) = [1,2]$, $\theta(\#A) = A1$, $\theta(\#X) = X1$ and $\theta(\#Z) = Z1$. The clause C_6 can be further transformed by the application of r_5. Notice that r_5 is also applicable to the clause C_2 of Section 2 at $\{append(X,Y,[1,2,3])\}$. □

Since the rule r_2 of Section 2 and the rule r_5 of Example 5.2 are applicable to an *initial*-atom of any pattern and an *append*-atom of any pattern, respectively, and their applications correspond to the unfolding operation, they will be referred to as *unfolding-based general* rewriting rules. The next example illustrates rewriting rules that are devised for atoms of specific patterns.

Example 5.3 Referring to the definition part D_{init} of Example 1.1, consider the query part consisting only of the clause C_4 of Example 5.2. Suppose that the rewriting rules prepared from the definition part D_{init} include the rules:

r_6: $append(\&X, [\&E], [\&A, \&B|\&Z])$
$\rightarrow equal(\&X, [\&A|\#W]), append(\#W, [\&E], [\&B|\&Z])$

r_7: $append(\&X, [\&E], [\&A]) \rightarrow equal(\&X, [\,]), equal(\&E, \&A)$

The rewriting rule r_6 can be applied to C_4 at $\{append(X, [E], [1, 2])\}$, transforming C_4 into the clause

C_7: $ans(X) \leftarrow equal(X, [1|W]), append(W, [E], [2])$.

Then, by applying the rule r_7 to C_7 at $\{append(W, [E], [2])\}$, C_7 can be transformed into the clause

C_8: $ans(X) \leftarrow equal(X, [1|W]), equal(W, [\,]), equal(E, 2)$,

from which the answer, $X = [1]$, can be derived. In comparison to the application of the rule r_5 in Example 5.2, notice that neither the application of r_6 nor that of r_7 increases the number of clauses in the query part. In general, the efficiency of computation can be improved by avoiding transformation steps that increase the number of clauses. □

Next, what it means for a rewriting rule to be correct is formally defined.

Correctness of Rewriting Rules
Let $D \in Dscr(R_1, R_1)$. A rewriting rule r on R_1 is *correct with respect to D and R_2*, if and only if for any declarative description $Q \in Dscr(R_1, R_2)$ and any definite clauses $C, C_1, \ldots, C_n$ from R_1 to R_2, if r rewrites C into $C_1, \ldots, C_n$, then

$$\mathcal{M}(D \cup Q \cup \{C\}) = \mathcal{M}(D \cup Q \cup \{C_1, \ldots, C_n\}).$$

6 Correctness of Reverse Rewriting Rules

Based on the established foundation for correctness of rewriting rules, it will now be shown that one can in general construct a correct rewriting rule by simply reversing another correct rewriting rule.

Theorem 6.1 (Correctness of Reverse Rewriting Rules)
Let $D \in Dscr(R_1, R_1)$. Let r be a rewriting rule

$$\hat{A}s \rightarrow \hat{B}s$$

on R_1. Let $reverse(r)$ be the rewriting rule

$$\hat{B}s \rightarrow \hat{A}s$$

on R_1. If r is correct with respect to D and R_2, then $reverse(r)$ is also correct with respect to D and R_2.

Proof.

Let Q be a declarative description in $Dscr(R_1, R_2)$, C a definite clause

$$C: \quad H \leftarrow Bs \cup Bs'$$

from R_1 to R_2, and let r be correct with respect to D and R_2. Suppose that $reverse(r)$ is applied to C at Bs by using a meta-variable instantiation θ. Then, $Bs = \hat{Bs}\theta$ and $reverse(r)$ rewrites C into the clause

$$C': \quad H \leftarrow \hat{As}\theta \cup Bs'.$$

It has to be shown that $\mathcal{M}(D \cup Q \cup \{C\}) = \mathcal{M}(D \cup Q \cup \{C'\})$. Clearly, by using the meta-variable instantiation θ, r is applicable to C' at the set $\hat{As}\theta$. This application of r rewrites the set $\hat{As}\theta$ in the body of C' into $\hat{Bs}\theta$, which is equal to Bs. That is, C' is rewritten into C by this application. Since r is correct with respect to D and R_2, $\mathcal{M}(D \cup Q \cup \{C\})$ and $\mathcal{M}(D \cup Q \cup \{C'\})$ are equal. So $reverse(r)$ is correct with respect to D and R_2. □

7 Conclusions

Each resolution step in the proof procedures associated with logic programming corresponds to an unfolding transformation step in RBET, which can be realized by the employment of unfolding-based general rewriting rules. However, while resolution is the only means of inference in logic programming, a variety of other rewriting rules can be used in RBET. The RBET framework therefore allows a wider variety of computation paths and, as a result, more efficient programs. Despite its simplicity, the RBET framework enables the development of a solid theoretical basis for determining the correctness of rewriting rules of various kinds. As long as correct rewriting rules are used throughout a transformation process, correct computation is always obtained. Experimental RBET-based knowledge processing systems in various application domains have been implemented at Hokkaido University, and satisfactory results revealing the usefulness of the framework have been obtained.

In this paper, the syntax for a large class of rewriting rules is proposed. This class of rewriting rules can represent unfolding-based general rewriting rules (e.g., the rules r_2 and r_5 of Subsection 2.1 and Example 5.2, respectively), folding-like rules (e.g., the rule r_4 of Subsection 2.2), and rules that are applicable to atoms of specific patterns (e.g., the rules r_6 and r_7 of Example 5.3). By incorporation of meta-variables of two kinds (&-variables and #-variables), the proposed syntax facilitates precise control of rewriting-rule instantiations and applications, which is necessary for ensuring the correctness of computation. A theoretical basis for verifying the correctness of rewriting rules is formulated. The reverse-transformation operation is introduced, and it is shown that in general a correct rewriting rule can be obtained by simply reversing another correct rewriting rule.

In addition to the necessity identified in this paper of the use of meta-

variables of the two kinds for specifying atom patterns in rewriting rules, it is demonstrated in [3] that the distinction between these two kinds of meta-variables also enables meaningful manipulation of atom patterns in the process of systematically generating rewriting rules from a definition part by means of meta-rules and is essential for controlling the generation process. Although reverse transformation may lead to an infinite loop in ordinary computation, it provides a foundation of folding-like meta-level transformation in the generation of rewriting rules and the correctness of reverse rewriting rules is essential for verifying the correctness of folding-like meta-rules.

Appendix

Referring to Example 1.1, Q can be transformed into Q' as follows. (The selected atom in each step is underlined.)

1: $ans(X) \leftarrow \underline{append(X, Y1, [1,2,3])}, initial(X, [1,3,5])$
2: $ans([\,]) \leftarrow \underline{initial([\,], [1,3,5])}$
$ans([1|X1]) \leftarrow append(X1, Y1, [2,3]), initial([1|X1], [1,3,5])$
3: $ans([\,]) \leftarrow \underline{append([\,], Y2, [1,3,5])}$
$ans([1|X1]) \leftarrow append(X1, Y1, [2,3]), initial([1|X1], [1,3,5])$
4: $ans([\,]) \leftarrow$
$ans([1|X1]) \leftarrow \underline{append(X1, Y1, [2,3])}, initial([1|X1], [1,3,5])$
5: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow \underline{initial([1], [1,3,5])}$
$ans([1|[2|X2]]) \leftarrow append(X2, Y1, [3]), initial([1|[2|X2]], [1,3,5])$
6: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow \underline{append([1], Y3, [1,3,5])}$
$ans([1|[2|X2]]) \leftarrow append(X2, Y1, [3]), initial([1|[2|X2]], [1,3,5])$
7: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow \underline{append([\,], Y3, [3,5])}$
$ans([1|[2|X2]]) \leftarrow append(X2, Y1, [3]), initial([1|[2|X2]], [1,3,5])$
8: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow$
$ans([1|[2|X2]]) \leftarrow append(X2, Y1, [3]), \underline{initial([1|[2|X2]], [1,3,5])}$
9: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow$
$ans([1|[2|X2]]) \leftarrow append(X2, Y1, [3]), \underline{append([1|[2|X2]], Y4, [1,3,5])}$
10: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow$
$ans([1|[2|X2]]) \leftarrow append(X2, Y1, [3]), \underline{append([2|X2], Y4, [3,5])}$
11: $ans([\,]) \leftarrow$
$ans([1]) \leftarrow$

There are several other possible ways of transforming Q into Q', some of which may result in a sequence that is shorter than the one shown above.

References

[1] Akama, K., Shigeta, Y., and Miyamoto, E., *Solving Problems by Equivalent Transformation of Logic Programs*, in Proceedings of the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99), Orlando, Florida, 1999.

[2] Akama, K., Kawaguchi, Y., and Miyamoto, E., *Equivalent Transformation for Equality Constraints on Multiset Domains* (in Japanese), Journal of the Japanese Society for Artificial Intelligence **13** (1998), pp. 395–403.

[3] Akama, K., Koike, H., and Miyamoto, E., *Program Synthesis from a Set of Definite Clauses and a Query*, in Proceedings of the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99), Orlando, Florida, 1999.

[4] Akama, K., Okada, K., and Miyamoto, E., *A Foundation of Equivalent Transformation of Negative Constraints on String Domains* (in Japanese), IEICE Technical Report, SS97-91, pp. 33–40, 1998.

[5] Lloyd, J. W., "Foundations of Logic Programming", second, extended edition, Springer-Verlag, 1987.

[6] Loveland, D. W. and Nadathur, G., *Proof Procedures for Logic Programming*, in: Gabbay, D. M., Hogger, C. J., and Robinson, J. A. (eds.), "Handbook of Logic in Artificial Intelligence and Logic Programming", Vol. 5, Oxford University Press, 1998, pp. 163–234.

[7] Nantajeewarawat, E., Akama, K., and Koike, H., *Expanding Transformation as a Basis for Correctness of Rewriting Rules*, in Proceedings of the Second International Conference on Intelligent Technologies (InTech'01), Bangkok, Thailand, 2001.

[8] Pettorossi, K. and Proietti, M., *Transformation of Logic Programs: Foundations and Techniques*, Journal of Logic Programming **19/20** (1994), pp. 261–320.

[9] Pettorossi, K. and Proietti, M., *Transformation of Logic Programs*, in: Gabbay, D. M., Hogger, C. J., and Robinson, J. A. (eds.), "Handbook of Logic in Artificial Intelligence and Logic Programming", Vol. 5, Oxford University Press, 1998, pp. 697–787.

[10] Robinson, J. A., *Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM **12** (1965), pp. 23–41.

Published in Proceedings of the Joint International Conference of SNLP - Oriental COCOSDA 2002, Hua Hin, Prachuapkirikhan, Thailand, pages 89-96, May 2002. ISBN 974-572-947-7.

The Roles of Ontologies in Manipulation of XML Data

Hataichanok Unphon Ekawit Nantajeewarawat

Information Technology Program
Sirindhorn International Institute of Technology, Thammasat University
P.O. BOX 22, Thammasat Rangsit Post Office, Pathumthani 12121, Thailand
e-mail : unphon@siit.tu.ac.th, ekawit@siit.tu.ac.th

Abstract

Generally defined as a formal specification of shared conceptualizations of a domain, an ontology provides a common understanding of topics that can be communicated between people and heterogeneous application systems. Limitations of Data Type Definitions and Resource Description Framework Schemas as ontology languages are identified; then, an ontology language, called $\mathcal{L}_{\text{ONTO}}$, is presented. As its distinctive features, $\mathcal{L}_{\text{ONTO}}$ separates class assertions clearly from class definitions and allows the inclusion of individuals in class expressions. The formal semantics of $\mathcal{L}_{\text{ONTO}}$ is provided by means of a translation into description logics and, alternatively, a translation into F-logic. Based-on these translations, the reasoning services provided by description logics as well as the resolution-based proof theory of F-logic can be applied for reasoning with ontologies and their instances.

1 Introduction

The Extensible Markup Language (XML) (Goldfarb and Prescod, 1998) has been widely known in the Internet community as a fundamental language that provides the underlying syntax of data for a rapidly growing number of Web-based applications and activities. XML itself, however, does not imply any interpretation of data; any intended semantics is outside the realm of XML specification (Decker et al., 2000; Klein, 2001). A qualitatively better level of XML-based automated information access and machine-understandable information provision necessitates additional explicit representation of the semantics of data and domain theories (Fensel and Musen, 2001; Fensel, 2001; Hendler, 2001).

The concept of ontology has been employed in knowledge engineering, natural language processing, and intelligent information integration as a formal, explicit specification of shared conceptualizations (i.e., meta-information) that describe the semantics of data (Uschold and Grunninger, 1996; Fensel, 2001). It has recently been adopted by the Semantic Web circles as a specification of a collection of knowledge terms, their semantic interconnections, some simple rules of inference, and logic for some particular domain (Hendler, 2001). In its simplest form, an ontology typically contains hierarchies of concepts (or classes) and describes properties of each concept through an attribute-value mechanism. It provides a common vocabulary for information exchange and a common understanding of a domain.

In this paper, limitations of using Data Type Definitions (DTDs) (Goldfarb and Prescod, 1998) and Resource Description Framework Schemas (RDF Schemas) (Brickley and Guha, 2000) for defining ontologies are identified (Section 2). Thereafter, an ontology language, i.e., a language for describing ontologies, called $\mathcal{L}_{\text{ONTO}}$, is presented (Section 3). $\mathcal{L}_{\text{ONTO}}$ provides a clear distinction between assertional perperties and definitional properties of individuals and allows the use of individuals in class expressions. The precise semantics of $\mathcal{L}_{\text{ONTO}}$ is defined by means of a translation into description logics (Borgida, 1995; Donini et al., 1996) (Section 4), and, alternatively, a translation into F-logic (Kifer et al., 1995) (Section 5). Through these translations, the inference mechanisms provided by description logics and F-logic can be employed for reasoning with ontologies and object-level XML data. Comparison of $\mathcal{L}_{\text{ONTO}}$ with the core language of the Web-based ontology infrastructure OIL (Decker et al., 2000; Fensel et al., 2001) is made (Section 6).

```
<SIIT-Student id="#088"
     first-name="Bhurit" last-name="Sittikul">
    <major>IT</major>
    <status>sohomore</status>
    <degree-prog>BSc</degree-prog>
    <taker>
        <course id="IT214"/>
        <course id="TU110"/>
    </taker>
</SIIT-Student>
```

Figure 1. A well-formed XML element.

```
<SIIT-Student id="#088">
    <name>Bhurit</name>
    <family-name>Sittikul</family-name>
    <department>IT</department>
    <status>second year</status>
    <degree-prog>BSc</degree-prog>
    <courses idrefs="IT214 TU110"/>
</SIIT-Student>
```

Figure 2. A well-formed XML element.

```
<!ELEMENT SIIT-Student
            (major, status, degree-prog, taker,)>
<!ELEMENT major (#PCDATA)>
<!-- major choices: CE, EE, IE, IT, ME -->
<!ELEMENT status (#PCDATA)>
<!-- status choices:
    freshy, sophomore, junior, senior, etc -->
<!ELEMENT degree-prog (#PCDATA)>
<!-- degree-prog choices: BSc, BEng, MSc, Phd -->
<!ELEMENT taker (course*)>
<!ELEMENT course EMPTY>
<!ATTLIST SIIT-Student
            id ID #REQUIRED
            first-name CDATA #IMPLIED
            last-name CDATA #IMPLIED>
<!ATTLIST course id ID #REQUIRED>
```

Figure 3. A simple DTD.

2 Limitations of DTDs and RDF Schemas

Different well-formed XML documents may provide the same information; for example, the well-formed XML elements in Figures 1 and 2 may equivalently describe a specific SIIT student. The parties that use XML for their data exchange must agree beforehand on the vocabulary (e.g., the names of elements and attributes) and its use. Such an agreement can be partly specified by a Data Type Definition (DTD) (Goldfarb and Prescod, 1998), which serves as a context-free grammar for XML documents. Using the DTD in Figure 3, for instance, the XML element in Figure 1 is valid, whereas that in Figure 2 is not.

A DTD, however, provides only a simple structure prescription; it only defines the legal lexical nestings of elements, their order, their possible attributes, and the locations where normal text is allowed. It does not serve as a machine-processable description of the semantics of XML data. In the DTD in Figure 3, for example, the possible values of a major-element, which imply the meaning of the tag major and its usage, can only be given as a comment[1], which is not machine-understandable. Consequently, content-based semantic constraints on information cannot be specified using DTDs; for example, one cannot assert that if the major of an undergraduate SIIT student is IT, then his/her degree program is necessarily BSc. Moreover, generalization relationship (subclass relationship) among XML elements, which is a fundamental abstraction mechanism for relating the elements semantically, cannot be described by a DTD.

Resource Description Framework Schemas (RDF Schemas) (Brickley and Guha, 2000) provide a mechanism for expressing the semantics of data through metadata descriptions. Their basic modeling primitives include *class* definitions and *subclass-of* statements (which together allow the construction of a generalization hierarchy of classes), *property* definitions along with *domain* and *range* statements (for restricting the possible combinations of properties and classes), and *type* statements (for declaring an object as an instance of a class). For example, the RDF Schema in Figure 4 asserts that taker is a property (attribute) of every instance of the class SIIT-Student and the values of this property must be instances of the class Course, and that SIIT-GradStudent is a subclass of SIIT-Student; accordingly, SIIT-GradStudent inherits the property taker and its range restriction from SIIT-Student.

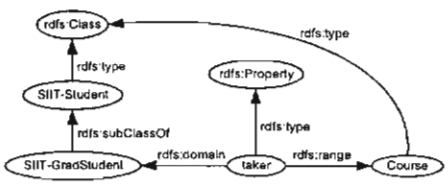

Figure 4. An RDF graph.

[1] A comment in a DTD is enclosed within a pair of <!-- and -->.

Nevertheless, properties are defined globally and are not encapsulated as attributes of classes in RDF Schemas (Fensel, 2001). As a consequence, the range restriction of a property of a certain subclass cannot be further refined. For example, one cannot provide an additional constraint that any value of the property taker of an instance of SIIT-GradStudent must be an instance of some specific subclass of Course, say AdvancedCourse. Such refinement of range restriction is apparently necessary for specifying semantic constraints for a subclass.

In RDF Schemas, only assertional properties of the instances of a class, i.e., necessary conditions for membership of the class, can be specified. There is no mechanism for providing an insight into the meaning of a class by specifying necessary and sufficient conditions for membership of the class. As a result, a class cannot be defined intensionally based on the properties of its instances; one cannot define, for example, SIIT-GradStudent as the class of all instances of SIIT-Student whose degree programs are either MSc or Phd. The distinction between assertions and definitions is important for a clearer understanding of the semantics of conceptual representation (Woods, 1991).

3 An ontology Language, $\mathcal{L}_{ONTO}$

This section presents an ontology language, which will be referred to as $\mathcal{L}_{ONTO}$. An ontology in $\mathcal{L}_{ONTO}$ is itself an XML document, consisting of slot declarations and class declarations. The syntax of $\mathcal{L}_{ONTO}$ is formally defined by the DTD in Figure 5.[2] For improvement of readability, a more compact pseudo XML syntax will be used, where opening tags are indicated by bold faced text, grouping of subcontents is indicated by indentation, and closing tags are omitted. Furthermore, the tag of a slotName-element and that of a className-element will be omitted, the content of a set-element will be written using the usual set notation, and the and-tag will be used as an infix operator.

An $\mathcal{L}_{ONTO}$ ontology is illustrated in Figure 6. It contains one slot declaration and four class declarations. The declaration of the slot *taker*

[2] Note that the DTD in Figure 5 does not define an ontology, but an ontology language, which is used to describe ontologies.

```
<!-- DTD for L_ONTO -->
<!ELEMENT ontology (slot*, class*)>

<!-- Slot Declaration -->
<!ELEMENT slot (slotName, domain, range)>
<!ELEMENT slotName CDATA #REQUIRED>
<!ELEMENT domain CDATA #REQUIRED>
<!ELEMENT range CDATA #REQUIRED>

<!-- Class Declaration -->
<!ELEMENT class (className, %property;)>
<!ELEMENT className CDATA #REQUIRED>

<!-- Class Property -->
<!ENTITY %property
      "((definition  assertion) | definition | assertion)">
<!ELEMENT definition (%constraint;)+>
<!ELEMENT assertion (%constraint;)+>

<!-- Constraint -->
<!ENTITY %constraint "(subclass-of | slot-constraint)">
<!ELEMENT subclass-of (className)>
<!ELEMENT slot-constraint (slotName,
      (value-type | unique-value-in | some-value-in))>
<!ELEMENT value-type (%classExpression;)>
<!ELEMENT unique-value-in (%classExpression;)>
<!ELEMENT some-value-in (%classExpression;)>

<!-- Class Expression -->
<!ENTITY %classExpression
      "(className | set | and | slot-constraint)">
<!ELEMENT  and (%classExpression;,
      (%classExpression;)+)>
<!ELEMENT set (li+)>
<!ELEMENT li CDATA #REQUIRED>
```

Figure 5. DTD specification for $\mathcal{L}_{ONTO}$.

simply asserts that if an individual x is related to an individual y by this slot relation, then x and y must be instances of the classes *Student* and *Course*, respectively. The declaration of a class contains an assertion part and a definition part, one of which may be omitted. The assertion part specifies necessary but not sufficient conditions for membership of the class; by contrast, the definition part provides necessary and sufficient conditions for the membership. Each of the two parts is a combination of *subclass-of* statements and *slot-constraint* statements. A *slot constraint* is a class expression that takes one of the three forms

1) R **value-type** E,
2) R **some-value-in** E,
3) R **unique-value-in** E,

where R is a slot name and E is a class expression. The slot constraints of the first, the second, and the third forms, respectively, denote

- the class consisting of every individual that is not related by R to any individual that is not an instance of the class denoted by E,

- the class consisting of every individual that is related by R to at least one instance of the class denoted by E (and is possibly also related by R to some individual of some other class),
- the class consisting of every individual that is related by R to exactly one individual in the class denoted by E and is not related by R to any other individual.

The first class declaration in Figure 6 asserts that every instance of *SIIT-Student* is necessarily

```
ontology
  slot taker
    domain Student
    range Course
  class SIIT-Student
    assertion
      subclass-of Student
      slot-constraint degree-prog
        unique-value-in
          {BSc, BEng, MSc, Phd}
      slot-constraint major
        unique-value-in {CE, EE, IE, IT, ME}
  class SIIT-Undergrad
    definition
      subclass-of SIIT-Student
      slot-constraint degree-prog
        value-type {BSc, BEng}
    assertion
      slot-constraint status
        unique-value-in
          {freshy, sophomore, junior, senior}
  class IT-Undergrad
    definition
      subclass-of SIIT-Undergrad
      slot-constraint major
        value-type {IT}
    assertion
      slot-constraint prog-lang
        some-value-in {C}
      slot-constraint prog-lang
        some-value-in {Java}
      slot-constraint degree-prog
        value-type {BSc}
  class SIIT-GradStudent
    definition
      subclass-of SIIT-Student
      slot-constraint degree-prog
        value-type {MSc, Phd}
    assertion
      slot-constraint taker
        value-type Course
          and slot-constraint lecturer
                value-type FullProfessor
```

Figure 6. An ontology in $\mathcal{L}_{\text{ONTO}}$.

an instance the class *Student*, the class denoted by the slot constraint (*degree-prog* **unique-value-in** {*BSc*, *BEng*, *MSc*, *Phd*}), and also the class denoted by the slot constraint (*major* **unique-value-in** {*CE*, *EE*, *IE*, *IT*, *ME*}); however, there may exist some individual that is an instance of each of these three classes but is not an instance of *SIIT-Student*. In plain words, this class declaration asserts that every SIIT student has a unique degree program, which is one of *BSc*, *BEng*, *MSc*, and *Phd*, and a unique major, which is one of *CE*, *EE*, *IE*, *IT*, and *ME*; but it does not provide the definition of an SIIT student. The next class declaration defines *SIIT-Undergrad* as the class consisting of every instance of *SIIT-Student* that is also an instance of the class denoted by the slot constraint (*degree-prog* **value-type** {*BSc*, *BEng*}). Intuitively, it defines an SIIT undergrad(uate) as an SIIT student whose degree program is either *BSc* or *BEng*. Then, it specifies as an assertion that every instance of *SIIT-Undergrad* is necessarily an instance of the class denoted by the slot constraint (*status* **unique-value-in** {*freshy*, *sophomore*, *junior*, *senior*}), but not vice versa. Likewise, the third and the fourth class declarations provide the definitions of the classes *IT-Undergrad* and *SIIT-GradStudent*, respectively, and describe some of their properties as assertions. Note that since a slot constraint is itself a class expression, it may be used to specify another slot constraint; the nested slot constraint in the last assertion part in Figure 6, for example, intuitively denotes the class consisting of every individual that takes no course that is not lectured by a full processor.

4 Translation into Description Logics

The formal semantics of $\mathcal{L}_{\text{ONTO}}$ will be defined in this section by means of a translation into a concept language in description logics.

4.1 Description Logics

Description logics (also called terminological logics) (Borgida, 1995; Donini et al., 1996) stem from Semantic Networks and Frames. They deal with the representation of structured concepts, their semantics and reasoning with them. The structure of a concept is described using a language, called concept language, comprising Boolean operators (conjunction, disjunction, ne-

$$
\begin{array}{rcl}
C, D & \rightarrow & A \mid \top \mid \bot \mid \neg A \mid C \sqcap D \\
 & & \mid \forall R.C \mid \exists R \mid \exists R.C \\
 & & \mid (\geq n\ R) \mid (\leq n\ R) \\
 & & \mid \{a_1, \ldots, a_n\}
\end{array}
$$

Figure 7. Syntax of $\mathcal{ALENO}$ concepts.

$$
\begin{array}{rcl}
\top^I & = & \Delta^I \\
\bot^I & = & \varnothing \\
(\neg A)^I & = & \Delta^I \setminus A^I \\
(C \sqcap D)^I & = & C^I \cap D^I \\
(\forall R.C)^I & = & \{d_1 \in \Delta^I \mid \forall d_2 : (d_1, d_2) \in R^I \Rightarrow d_2 \in C^I\} \\
(\exists R)^I & = & \{d_1 \in \Delta^I \mid \exists d_2 : (d_1, d_2) \in R^I\} \\
(\exists R.C)^I & = & \{d_1 \in \Delta^I \mid \exists d_2 : (d_1, d_2) \in R^I \wedge d_2 \in C^I\} \\
(\geq n\ R)^I & = & \{d_1 \in \Delta^I \mid \#\{d_2 \mid (d_1, d_2) \in R^I\} \geq n\} \\
(\leq n\ R)^I & = & \{d_1 \in \Delta^I \mid \#\{d_2 \mid (d_1, d_2) \in R^I\} \leq n\} \\
\{a_1, \ldots, a_n\}^I & = & \{a_1{}^I, \ldots, a_n{}^I\}
\end{array}
$$

Figure 8. Conditions for an interpretation I.

gation) and various forms of quantification over the roles (also called attributes or slots) of the concept. The language $\mathcal{ALENO}$ in the commonly known family of $\mathcal{AL}$-languages (Donini et al., 1996; Schaerf, 1994) will be used as the target concept language in this paper. Given an alphabet $\mathcal{P}$ of primitive concepts, an alphabet $\mathcal{R}$ of roles and an alphabet O of individuals, a *concept* in $\mathcal{ALENO}$ is constructed by means of the syntax rule in Figure 7, where C and D denote concepts, and A, R and the a_i belong to the alphabets $\mathcal{P}$, $\mathcal{R}$ and O, respectively.

An interpretation $I = (\Delta^I, \cdot^I)$ consists of a nonempty set Δ^I (the *domain* of I) and a function $\cdot^I$ (the *interpretation function* of I) that maps every concept to a subset of Δ^I, every role to a subset of $\Delta^I \times \Delta^I$ and every individual to an element of Δ^I such that the equations in Figure 8 are all satisfied. In addition, it is assumed that different individuals denote different elements in Δ^I (Unique Name Assumption), i.e., for any pair of individuals $a, b \in O$, if $a \neq b$, then $a^I \neq b^I$.

An interpretation I is a *model* for a concept C if C^I is nonempty. A concept is *satisfiable* if it has a model, and *unsatisfiable* otherwise.

A knowledge base built using description logics consists of two components: the *intensional* one, called *T-box*, and the *extensional* one, called *A-box*. A statement in a T-box has either the form $C \sqsubseteq D$ or the form $C \doteq D$, where C and D are concepts. An interpretation I *satisfies* the statement $C \sqsubseteq D$ if $C^I \subseteq D^I$, and the statement $C \doteq D$ if $C^I = D^I$. An interpretation I is a *model* for a T-box $\mathcal{T}$ if I satisfies every statement in $\mathcal{T}$.

A statement in an A-box takes either the form $C(a)$ or $R(a, b)$, where C is a concept, R is a role, and a, b are individuals. An interpretation I *satisfies* the statement $C(a)$ if $a^I \in C^I$, and the statement $R(a, b)$ if $(a^I, b^I) \in R^I$. An interpretation I is a *model* for an A-box $\mathcal{A}$ if I satisfies every statement in $\mathcal{A}$.

An interpretation I is a *model* for a knowledge base $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$ is a T-box and $\mathcal{A}$ is an A-box, if I is both a model for a $\mathcal{T}$ and a model for $\mathcal{A}$. A knowledge base Σ *logically implies* a statement α, written as $\Sigma \models \alpha$, if every model of Σ satisfies α.

$$
\begin{array}{rcl}
\sigma(\textbf{ontology } slotDecls\ classDecls) & = & \sigma(slotDecls) \cup \sigma(classDecls) \\
\sigma(slotDecl_1 \ldots slotDecl_n) & = & \sigma(slotDecl_1) \cup \ldots \cup \sigma(slotDecl_n) \\
\sigma(classDecl_1 \ldots classDecl_n) & = & \sigma(classDecl_1) \cup \ldots \cup \sigma(classDecl_n) \\
\sigma(\textbf{slot } R \textbf{ domain } A \textbf{ range } B) & = & \{(\exists R.\top \sqsubseteq \sigma(A)), (\top \sqsubseteq \forall R.\sigma(B))\} \\
\sigma(\textbf{class } A \textbf{ definition } constraints) & = & \{(A \doteq \top \sqcap \sigma(constrains))\} \\
\sigma(\textbf{class } A \textbf{ assertion } constraints) & = & \{(A \sqsubseteq \top \sqcap \sigma(constrains))\} \\
\sigma(\textbf{class } A \textbf{ definition } constraints_1 \textbf{ assertion } constraints_2) & = & \sigma(\textbf{class } A \textbf{ definition } constraints_1) \cup \\
 & & \sigma(\textbf{class } A \textbf{ assertion } constraints_2) \\
\sigma(subclConstr_1 \ldots subclConstr_n\ slotConstr_1 \ldots slotConstr_m) & = & \sigma(subclConstr_1) \sqcap \ldots \sqcap \sigma(subclConstr_n) \\
 & & \sqcap \sigma(slotConstr_1) \sqcap \ldots \sqcap \sigma(slotConstr_m) \\
\sigma(\textbf{subclass-of } A) & = & A \\
\sigma(\textbf{slot-constraint } R \textbf{ value-type } classExpr) & = & (\forall R.\sigma(classExpr)) \\
\sigma(\textbf{slot-constraint } R \textbf{ unique-value-in } classExpr) & = & (\exists R.\sigma(classExpr) \sqcap (\leq 1\ R)) \\
\sigma(\textbf{slot-constraint } R \textbf{ some-value-in } classExpr) & = & (\exists R.\sigma(classExpr)) \\
\sigma(classExpr_1 \textbf{ and } \ldots \textbf{ and } classExpr_n) & = & (\sigma(classExpr_1) \sqcap \ldots \sqcap \sigma(classExpr_n)) \\
\sigma(A) & = & A \\
\sigma(\{a_1, \ldots, a_n\}) & = & \{a_1, \ldots, a_n\}
\end{array}
$$

Figure 9. Translation of $\mathcal{L}_{\text{ONTO}}$ into $\mathcal{ALENO}$.

4.2 Translation of $\mathcal{L}_{\text{ONTO}}$ into $\mathcal{ALENO}$

A translation σ that maps ontologies in $\mathcal{L}_{\text{ONTO}}$ into T-boxes in $\mathcal{ALENO}$ is defined in Figure 9, where A , B denote class names and R denotes a slot name. As an illustration, by using the translation σ, the ontology in Figure 6 is transformed into a T-box consisting of the statements in Figure 10. While class declarations and slot declarations in an ontology is transformed into T-box statements, object-level XML elements (XML elements describing specific objects) will be transformed in a straightforward way into statements in an A-box. For example, the XML element in Figure 1 is converted into the A-box statements in Figure 11.

To demonstrate reasoning with ontologies and object-level XML data based on description logics, assume that $\mathcal{T}$ is a T-box consisting of the statements in Figure 10, $\mathcal{A}$ is an A-box containing the statements in Figure 11, and Σ is the knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$. Now let $I = (\Delta^I, \cdot^I)$ be a model for Σ. From the third and the fourth statements in Figure 10, I necessarily satisfies the statement *SIIT-Undergrad*(*#088*). Hence,

$$\Sigma \models \textit{SIIT-Undergrad}(\textit{\#088}),$$

i.e., the implicit information that the individual *#088* belongs to the class *SIIT-Undergrad* can be derived. Then, from the third and the sixth T-box statements in Figure 10, it is readily seen that

$$\Sigma \models \textit{IT-Undergrad}(\textit{\#088}).$$

Next, it follows from the seventh statement in Figure 10 that the model I necessarily satisfies the statements *prog-lang*(*#088*, *C*) and *prog-lang*(*#088*, *Java*). Therefore,

$$\Sigma \models \textit{prog-lang}(\textit{\#088}, C),$$
$$\Sigma \models \textit{prog-lang}(\textit{\#088}, \textit{Java}).$$

Consequently, the elements

```
<prog-lang>C</prog-lang>
<prog-lang>Java</prog-lang>
```

are both derived as implicit subelements of the SIIT-Student-element in Figure 1.

Besides derivation of implicit information, the framework of description logics also facilitates content-based validation of object-level XML data with respect to a given ontology For instance, suppose that the status-subelement in Figure 1 is replaced with the element

1. $(\exists taker.\top \sqsubseteq Student)$
2. $(\top \sqsubseteq \forall taker.Course)$
3. $(SIIT\text{-}Student \doteq \top \sqcap Student \sqcap (\exists degree\text{-}prog.\{BSc, BEng, MSc, Phd\} \sqcap (\leq 1\ degree\text{-}prog)) \sqcap (\exists major.\{CE, EE, IE, IT, ME\} \sqcap (\leq 1\ major)))$
4. $(SIIT\text{-}Undergrad \doteq \top \sqcap SIIT\text{-}Student \sqcap (\forall degree\text{-}prog.\{BSc, BEng\}))$
5. $(SIIT\text{-}Undergrad \sqsubseteq \top \sqcap (\exists status.\{freshy, sophomore, junior, senior\} \sqcap (\leq 1\ status)))$
6. $(IT\text{-}Undergrad \doteq \top \sqcap SIIT\text{-}Undergrad \sqcap (\forall major.\{IT\}))$
7. $(IT\text{-}Undergrad \sqsubseteq \top \sqcap (\exists prog\text{-}lang.\{C\}) \sqcap (\exists prog\text{-}lang.\{Java\}) \sqcap (\forall degree\text{-}prog.\{BSc\}))$
8. $(SIIT\text{-}GradStudent \doteq \top \sqcap SIIT\text{-}Student \sqcap (\forall degree\text{-}prog.\{MSc, Phd\}))$
9. $(SIIT\text{-}GradStudent \sqsubseteq \top \sqcap (\forall taker.(Course \sqcap \forall lecturer.FullProfessor)))$

Figure 10. Resulting T-box statements.

SIIT-Student(*#088*)	*first-name*(*#088*, *Bhurit*)
last-name(*#088*, *Sittikul*)	*major*(*#088*, *IT*)
status(*#088*, *sophomore*)	*degree-prog*(*#088*, *BSc*)
taker(*#088*, *IT214*)	*Course*(*IT214*)
taker(*#088*, *TU110*)	*Course*(*TU110*)

Figure 11. A-box statements.

```
<status>single</status>,
```

and, accordingly, the statement *status*(*#088*, *sophomore*) in the A-box $\mathcal{A}$ is replaced with

status(*#088*, *single*).

Since Σ logically implies the statement *SIIT-Undergrad*(*#088*), it follows that every interpretation that satisfies the statement *status*(*#088*, *single*) does not satisfy the fifth statement in Figure 10. As a result, the replacement leads to the inexistence of any model for Σ, which indicates that Σ becomes inconsistent. This inconsistency reflects the fact that the resulting XML element does not conform to the ontology in Figure 6, which asserts as a necessary condition that the status of an individual of the class *SIIT-Undergrad* can only be *freshy*, *sophomore*, *junior*, or *senior*.

5 Translation into F-Logic

Alternatively, the semantics of $\mathcal{L}_{\text{ONTO}}$ can be defined by means of a translation into a subclass of

F-logic (Kifer et al., 1995)—a full-fledged logic that has been widely recognized as a well-established theoretical foundation for the object-oriented paradigm. After identifying the subclass considered in this paper of F-logic, such a translation will be presented in this section.

Given an alphabet O of individuals, an alphabet C of class names, an alphabet $\mathcal{R}$ of attribute names and an alphabet $\mathcal{V}$ of variables, an *F-logic atomic formula* (*F-atom*) used in this paper takes one of the three forms

1) *id-term*:*A*,
2) $A[R \Rightarrow\!> B]$,
3) *id-term*$[R \rightarrow\!> \textit{id-term}']$,

where *id-term* and *id-term'* are elements of $O \cup \mathcal{V}$, A and B belong to C, and R belongs to $\mathcal{R}$. A ground (variable-free) F-atom of the first form is intended to mean "the object denoted by *id-term* is an instance of the class A", that of the second form is intended to mean "each value of the attribute R of an instance of the class A is necessarily an instance of the class B", and that of the third form is intended to mean "the object denoted by *id-term'* is a value the attribute R of the object denoted by *id-term*". F-logic statements, called *F-formulas*, are constructed inductively out of F-atoms by means of standard logical connectives and quantifiers in the usual way:

- F-atoms are F-formulas;
- If φ and ψ are F-formulas, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$ are F-formulas;
- If φ and ψ are F-formulas and x, $y \in \mathcal{V}$, then $\forall x(\varphi)$ and $\exists y(\psi)$ are F-formulas.

$$
\begin{aligned}
\rho(\textbf{ontology } \textit{slotDecls} \ldots \textit{classDecls}) &= \rho(\textit{slotDecls}) \cup \rho(\textit{classDecls}) \\
\rho(\textit{slotDecl}_1 \ldots \textit{slotDecl}_n) &= \rho(\textit{slotDecl}_1) \cup \ldots \cup \rho(\textit{slotDecl}_n) \\
\rho(\textit{classDecl}_1 \ldots \textit{classDecl}_n) &= \rho(\textit{classDecl}_1) \cup \ldots \cup \rho(\textit{classDecl}_n) \\
\rho(\textbf{slot } R \textbf{ domain } A \textbf{ range } B) &= A[R \Rightarrow\!> B] \\
\rho(\textbf{class } A \textbf{ definition } \textit{constraints}) &= \{\forall x_1 (x_1 : A \Leftrightarrow \rho(1, \textit{constraints}))\} \\
\rho(\textbf{class } A \textbf{ assertion } \textit{constraints}) &= \{\forall x_1 (x_1 : A \Rightarrow \rho(1, \textit{constraints}))\} \\
\rho(\textbf{class } A \textbf{ definition } \textit{constraints}_1 \textbf{ assertion } \textit{constraints}_2) &= \rho(\textbf{class } A \textbf{ definition } \textit{constraints}_1) \\
&\quad \cup \rho(\textbf{class } A \textbf{ assertion } \textit{constraints}_2) \\
\rho(i, \textit{subclConstr}_1 \ldots \textit{subclConstr}_n \textit{ slotConstr}_1 \ldots \textit{slotConstr}_m) &= (\rho(i, \textit{subclConstr}_1) \wedge \ldots \wedge \rho(i, \textit{subclConstr}_n) \\
&\quad \wedge \rho(I, \textit{slotConstr}_1) \wedge \ldots \wedge \rho(i, \textit{slotConstr}_m)) \\
\rho(i, \textbf{subclass-of } A) &= (x_i : A) \\
\rho(i, \textbf{slot-constraint } R \textbf{ value-type } \textit{classExpr}) &= \forall x_{i+1}(x_i[R \rightarrow\!> x_{i+1}] \Rightarrow \rho(i{+}1, \textit{classExpr})) \\
\rho(i, \textbf{slot-constraint } R \textbf{ unique-value-in } \textit{classExpr}) &= \exists x_{i+1}(x_i[R \rightarrow\!> x_{i+1}] \wedge \forall y(x_i[R \rightarrow\!> y] \Rightarrow y = x_{i+1}) \\
&\quad \wedge \rho(i{+}1, \textit{classExpr})) \\
\rho(i, \textbf{slot-constraint } R \textbf{ some-value-in } \textit{classExpr}) &= \exists x_{i+1}(x_i[R \rightarrow\!> x_{i+1}] \wedge \rho(i{+}1, \textit{classExpr})) \\
\rho(i, \textit{classExpr}_1 \textbf{ and } \ldots \textbf{ and } \textit{classExpr}_n) &= (\rho(i, \textit{classExpr}_1) \wedge \ldots \wedge \rho(i, \textit{classExpr}_n)) \\
\rho(i, A) &= (x_i{:}A) \\
\rho(i, \{a_1, \ldots, a_n\}) &= (x_i = a_1 \vee \ldots \vee x_i = a_n)
\end{aligned}
$$

Figure 12. Translation of $\mathcal{L}_{\text{ONTO}}$ into F-Logic.

Student$[taker \Rightarrow\!> Course]$
$\forall x_1(x_1{:}\textit{SIIT-Student} \Rightarrow ((x_1{:}\textit{Student})$
$\wedge \exists x_2(x_1[\textit{degree-prog} \rightarrow\!> x_2] \wedge \forall y(x_1[\textit{degree-prog} \rightarrow\!> y] \Rightarrow y = x_2) \wedge (x_2 = \textit{BSc} \vee x_2 = \textit{BEng} \vee x_2 = \textit{MSc} \vee x_2 = \textit{Phd}))$
$\wedge \exists x_2(x_1[\textit{major} \rightarrow\!> x_2] \wedge \forall y(x_1[\textit{major} \rightarrow\!> y] \Rightarrow y = x_2) \wedge (x_2 = \textit{CE} \vee x_2 = \textit{EE} \vee x_2 = \textit{IE} \vee x_2 = \textit{IT} \vee x_2 = \textit{ME}))))$
$\forall x_1(x_1{:}\textit{SIIT-Undergrad} \Leftrightarrow ((x_1{:}\textit{SIIT-Student}) \wedge \forall x_2(x_1[\textit{degree-prog} \rightarrow\!> x_2] \Rightarrow (x_2 = \textit{BSc} \vee x_2 = \textit{BEng}))))$
$\forall x_1(x_1{:}\textit{SIIT-Undergrad} \Rightarrow (\exists x_2(x_1[\textit{status} \rightarrow\!> x_2] \wedge \forall y(x_1[\textit{status} \rightarrow\!> y] \Rightarrow y = x_2)$
$\wedge (x_2 = \textit{freshy} \vee x_2 = \textit{sophomore} \vee x_2 = \textit{junior} \vee x_2 = \textit{senior}))))$
$\forall x_1(x_1{:}\textit{IT-Undergrad} \Leftrightarrow ((x_1{:}\textit{SIIT-Undergrad}) \wedge \forall x_2(x_1[\textit{major} \rightarrow\!> x_2] \Rightarrow (x_2 = \textit{IT}))))$
$\forall x_1(x_1{:}\textit{IT-Undergrad} \Rightarrow (\exists x_2(x_1[\textit{prog-lang} \rightarrow\!> x_2] \wedge (x_2 = C)) \wedge \exists x_2(x_1[\textit{prog-lang} \rightarrow\!> x_2] \wedge (x_2 = \textit{Java}))$
$\wedge \forall x_2(x_1[\textit{degree-prog} \rightarrow\!> x_2] \Rightarrow (x_2 = \textit{BSc}))))$
$\forall x_1(x_1{:}\textit{SIIT-GradStudent} \Leftrightarrow ((x_1{:}\textit{SIIT-Student}) \wedge \forall x_2(x_1[\textit{degree-prog} \rightarrow\!> x_2] \Rightarrow (x_2 = \textit{MSc} \vee x_2 = \textit{Phd}))))$
$\forall x_1(x_1{:}\textit{SIIT-GradStudent} \Rightarrow (\forall x_2(x_1[\textit{taker} \rightarrow\!> x_2] \Rightarrow ((x_2{:}\textit{Course}) \wedge \forall x_3(x_2[\textit{lecturer} \rightarrow\!> x_3] \Rightarrow (x_3{:}\textit{FullProfessor}))))))$

Figure 13. Resulting statements in F-logic.

#088:SIIT-Student	*#088[first-name ->> Bhurit]*
#088[last-name ->> Sittikul]	*#088[major ->> IT]*
#088[status ->> sophomore]	*#088[degree-prog ->> BSc]*
#088[taker ->> IT214]	*IT214:Course*
#088[taker ->> TU110]	*TU110:Course*

Figure 14. Object-level F-formulas.

The reader is referred to (Kifer et al., 1995) for the formal semantics of F-logic.

Figure 12 defines a mapping ρ for translating an $\mathcal{L}_{\text{ONTO}}$ ontology into a set of F-formulas. Through ρ, the ontology in Figure 6 is transformed into a set consisting of F-formulas in Figure 13. Together with the transformation of a given ontology, object-level XML elements can also be translated in a direct way into F-formulas; for example, the F-formulas obtained from the XML element in Figure 1 are shown in Figure 14. By means of the mapping ρ, the model-theoretic semantics and the resolution-based proof theory of F-logic, which are elaborated in (Kifer et al., 1995), can be employed for reasoning with ontologies and object-level XML elements.

6 Related Work

In their collaborative proposals, Decker et al. (2000) and Fensel et al. (2001) enriched RDF Schemas with additional modeling primitives into an influential Web-based onlotogy infrastructure called Ontology Inference Layer (OIL), which partly inspires the work presented in this paper. OIL provides a core ontology language, in comparison with which $\mathcal{L}_{\text{ONTO}}$ has two distinctive features: a clearer distinction between class definitions and class assertions, and the inclusion of individuals in class expressions. In the core language of OIL, necessary but insufficient conditions for class membership can only be specified for a primitive class but not for a defined class, and the use of individuals in specifying slot values or defining extensional class expressions (i.e., class expressions defined by enumerating individuals) is not allowed. The core language of OIL, however, provides richer modeling primitives for specifying global constraints that apply to slot relations, e.g., a slot relation can be specified to be transitive, symmetric, or an inverse of another slot relation.

Acknowledgement

This work was supported by the Thailand Research Fund, under Grant No. PDF/31/2543.

References

Borgida, A., Description Logics in Data Management, *IEEE Transcations on Knowledge and Data Engineering*, 7(5): 671–682, 1995.

Brickley, D. and Guha, R. V., Resource Description Framework (RDF) Schema Specification 1.0, http://www.w3c.org/TR/2000, 2000.

Decker S., Melnik, S., van Harmelem, F., Fensel, D., Klein M., Broekstra, J., Erdman, M., and Horrocks, I. The Semantic Web: The Roles of XML and RDF, *IEEE Internet computing*, 4(5): 63–74, 2000.

Donini, F., Lenzerini, M., Nardi, D., and Schaerf A., Reasoning in Description Logics, in Brewka, G., editor, *Principles of Knowledge Representation and Reasoning*, CLSI Publication, pp. 193–238, 1996.

Fensel, D., *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer-Verlag, 2001.

Fensel, D. and Musen, M. A., The Semantic Web: A Brain of Humankind, *IEEE Intelligent Systems*, 16(2): 24–25, 2001.

Fensel, D., van Harmelen, F., Harrocks, I., McGuinness, D. L., and Petel-Scheider, P. F., OIL: An Ontology Infrastructure for the Semantic Web, *IEEE Intelligent Systems*, 16(2): 38–45, 2001.

Goldfarb, C. F. and Prescod, P., *The XML Handbook*, Prentice Hall, 1998.

Hendler, J., Agents and the Semantic Web, *IEEE Intelligent Systems*, 16(2): 30–37, 2001.

Kifer, M., Lausen, G., and Wu, J., Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of Association of Computing Machinery*, 42(4): 741–843, 1995.

Klein, M., XML, RDF, and Relatives, *IEEE Intelligent Systems*, 16(2): 26–28, 2001.

Schaerf, A., Reasoning with Individuals in Concept Languages, *Data and Knowledge Engineering*, 13(2): 141–176, 1994.

Uschold, M. and Gruninger, M., Ontologies: Principles, Methods and Applicaions, *Knowledge Engineering Review*, 11(2): 93–136, 1996.

Woods, W. A., Understanding Subsumption and Taxonomy: A Framework for Progress, in Sowa, J., editor, *Principles of Semantic Networks*, Morgan Kaufman Publishers, 1991.