Journal of Intelligent Information Systems, 1, 243–270 (1992)

Supporting Inconsistent Rules in Database Systems*

YANNIS E. IOANNIDIS†
Computer Sciences Department, University of Wisconsin, Madison, WI 53706

TIMOS K. SELLIS‡
Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742

Abstract. When a set of rules generates (conflicting) values for a virtual attribute of some tuple, the system must resolve the inconsistency and decide on a unique value that is assigned to that attribute. In most current systems, the conflict is resolved based on criteria that choose one of the rules in the conflicting set and use the value that it generated. There are several applications, however, where inconsistencies of the above form arise, whose semantics demand a different form of resolution. We propose a general framework for the study of the conflict resolution problem, and suggest a variety of resolution criteria, which collectively subsume all previously known solutions. With several new criteria being introduced, the semantics of several applications are captured more accurately than in the past. We discuss how conflict resolution criteria can be specified at the schema or the rule-module level. Finally, we suggest some implementation techniques based on rule indexing, which allow conflicts to be resolved efficiently at compile time, so that at run time only a single rule is processed.

Keywords: deductive database systems, inconsistent rules, rule compilation, rule indexing, virtual attributes

1. Introduction

Any organization or enterprise has its own unique structure and policies that determine its daily function. Consider one such organization and let the following express its policy with respect to the dress code of its employees:

r_1 : All employees who work on the first floor wear red shirts.
r_2 : All employees who work on the second floor wear green shirts.

* An earlier version of this work appeared under the title "Conflict Resolution of Rules Assigning Values to Virtual Attributes" in *Proceedings of the 1989 ACM-Sigmod Conference*, Portland, OR, June 1989, pp. 205–214.

† Partially supported by the National Science Foundation under Grant IRI-9157368 (PYI Award) and by grants from DEC, HP, and AT&T.

‡ Partially supported by the National Science Foundation under Grant IRI-9057573 (PYI Award), IBM, DEC, and the University of Maryland Institute for Advanced Computer Studies (UMIACS).

r_3 : The Red-Pants department is located on the first floor.
r_4 : The Green-Pants department is located on the second floor.

Each one of r_1 to r_4 is a *rule* that captures some aspect of the overall dressing policy of the company. Arguably, the use of rules for such purposes is very natural and intuitive. Such rules (although probably not those of dress code) are as important to an organization as the data that it has collected and maintains, and any decision made is usually based on both types of information. Thus, it is very important to be able to integrate rules and data into one *knowledge base* consisting of a *rulebase* and a *database* managed by a single system. Mathematical logic, and primarily first order logic, is commonly used as a language to express such rules. Systems that support such languages and also manage large disk-oriented databases are called *deductive database systems.*

One of the main advantages of using rules in the above capacity is that knowledge evolution is rather straightforward and does not require any major reorganization of the knowledge base. Rules can be added or deleted without much further action by the rulebase designer. The unfortunate consequence of this, however, is that the rulebase may soon become *inconsistent*, i.e., it may become possible to infer contradictory facts from the rules. The rules in the above example happened to be consistent. For example, assuming that John Red works in the Red-Pants department, when the query "What is the color of the shirt that John Red wears?" is posed, there is only one answer that can be inferred: "red." Consider the addition of the following rule to the above set:

r_5 : All supervisors wear blue shirts.

Clearly, the new rule set in inconsistent, since it cannot uniquely infer the color of the shirt of Jane Blue who is the supervisor of the Red-Pants department.

Assuming a large rulebase, it is impossible for the rulebase designer to guarantee the consistency of the rules. In fact, the rules of most organizations do contain inconsistencies, which whenever revealed are handled by *meta-rules* established for that purpose. It is desirable that deductive database systems follow the same approach as well. The rulebase designer should specify meta-rules to resolve inconsistencies which the system invokes whenever needed to produce only consistent results. How this can be done in a deductive database system is the general theme of this work.

In this paper, we are interested in a special case of rules that can lead to inconsistent inferences, rules that assign values to attributes of relations. Such attributes, whose values are not explicitly stored in the database, but are inferred by rules, are called *virtual attributes.* For example, in the above database, "shirt_color" might be a virtual attribute of the employee relation. Whenever a virtual attribute of some tuple is referenced in a query, the system should return a unique value for it. If the rules that define the values for the attribute are inconsistent, they may collectively generated multiple values. The problem of

deciding on a unique value, taking into account all those generated, is the focus of this paper.

Based on the semantics of the virtual attributes in the above examples, it is clear that eventually one of the generated values should be selected as the true value of the attribute. Inconsistencies, however, arise in other types of applications as well, where the semantics of the virtual attributes require that the generated values are all *combined* in some way to derive a possibly new value for the attribute. For example, consider a rulebase that contains rules that determine whether the Space Shuttle should be launched or not, based on a variety of conditions. It is conceivable that different rules may give different answers in any specific situation. The desired semantics for resolving the conflict may be that the decision suggested by the majority of rules be adopted, or most likely that only if all rules decide positively for a launch, the Shuttle will indeed depart. In the latter case, the function that combines all generated values is, in some sense, the logical AND operation. This work addresses inconsistency resolution for both types of attribute semantics.

The paper is organized as follows. Section 2 formally defines the rule inconsistency problem and describes current solutions. Section 3 introduces a general framework in which several classes of conflict resolution criteria can be expressed. Section 4 presents several application examples paired up with conflict resolution schemes that capture the semantics of the corresponding application. Section 5 discusses how conflict resolution criteria can be specified at the schema or the rule-module level. Section 6 studies some implementation techniques based on rule indexing, which allow conflicts to be resolved efficiently at compile time. Finally, Section 7 summarizes the basic contributions of this work and discusses some directions for future work.

2. Problem formulation

In this section, we elaborate on the problem of inconsistencies in rules stored in a database system. We define the types of rules considered and we analyze the situations where inconsistencies arise. Finally, we give a brief summary of the solutions used in existing systems.

2.1. Rule model

Consider a fixed, possibly infinite, set C. A database D is a vector $D = (C_D, \mathbf{Q}_1, \ldots, \mathbf{Q}_n)$[1], where $C_D \subseteq C$ is a (possibly infinite) set, and for each $1 \leq i \leq n$, $\mathbf{Q}_i \subseteq C_D^{a_i}$ is a *relation of arity* a_i. We allow infinite relations in D so that primitive relations (e.g., $=, \geq$) and functions (e.g., addition) can be included in our model. Clearly, such relations and functions are directly evaluable and are not explicitly stored, Each element of $\mathbf{Q}_i$, is called a *tuple*. Elements of

relations constructed by combination of database relations are called tuples as well. Without loss of generality, we assume for simplicity that the constants in the database are typeless. Extending our ideas to a typed system is straightforward. In what follows, we use lowercase x and y to denote tuple variables in rules and uppercase X and Y to denote specific tuples that can be bound to x and y, respectively. The tuple formed by concatenating X and Y is denoted by $\langle X, Y \rangle$. (Similarly for concatenating two tuple variables x and y.) For any function g defined on tuples of the same arity as $\langle X, Y \rangle$, $g(x, y)$ denotes the function itself, whereas $g(X, Y)$ denotes the value returned by the function when applied on X and Y.

We consider rules that are equivalent to Horn clauses, i.e., they are of the form

$$\mathbf{Q}_1(x^{(1)}) \wedge \cdots \wedge \mathbf{Q}_k(x^{(k)}) \rightarrow \mathbf{Q}_0(x^{(0)}), \tag{1}$$

where for each i, $x^{(i)}$ is a vector of variables, constants, and functions of such. We assume that the Horn clauses are range-restricted, i.e., every variable that appears in the consequent appears in the antecedent also, under some nonprimitive (i.e., explicitly stored, finite) relation. The following are two examples of Horn clauses:

$$\mathbf{EMP}(name,sal,age,dept,num\_kids) \wedge sal > 50K \wedge age < 30 \rightarrow \mathbf{WELL\_PAID}(name,sal). \tag{2}$$

$$\mathbf{EMP}(name,sal,age,dept,num\_kids) \wedge dept = \text{"toy"} \rightarrow num\_kids = 0. \tag{3}$$

The relation in the consequent of (2.3) is "=". Formally, we should have written "=(*num_kids*,0)," but we use the infix notation for convenience. The same convention is used for the primitive literals in the antecedents.

There are several semantics that can be used for a set of rules $\{r_i\}$ with respect to the relations of a database D. One such semantics, which is our starting point, assumes that the contents of all nonprimitive relations are explicitly stored in the database. Primitive relations are directly available, so their full extent is known to the database system as well. With this semantics, rules are treated as *integrity constraints*, i.e., the database contents have to satisfy them at all times. This implies that assigning constants to the variables of a rule should either make the antecedent false or make the consequent true.

The above is not a very useful semantics in deductive database systems. Explicitly storing the contents of all nonprimitive relations in the database is undesirable. A better semantics treats some of the rules as *derivation rules*. The contents of database relations, called *intentional (virtual) relations*, can be implicitly derived by rules having those relations in their consequents. In this case, the so called *least fixpoint semantics* (VanEmden and Kowalski, 1976; Aho and Ullman, 1979) is used, i.e., the derived contents of each relation constitute the minimum set (minimum with respect to $\subseteq$) that satisfies all the rules. The

existence of such a minimum is guaranteed by the fact that only Horn clauses are considered (Tarski, 1955; Aho and Ullman, 1979).

Treating a rule as an integrity constraint or a derivation rule is not an inherent property of the rule. Some general guidelines on what the natural role of a rule is, state that rules with a user-defined (nonprimitive) relation in their consequent serve better as derivation rules, whereas those with a primitive relation in their consequent serve better as integrity constraints (Nicolas and Gallaire, 1978). According to those guidelines, (2) should be used as a derivation rule and (3) should be used as an integrity constraint. This, however, is more restrictive than necessary, since storing the contents of all attributes in a nonprimitive relation is often undesirable. By treating a rule whose consequent is of the form "variable = expression" as a derivation rule (like (3)), we are able to implicitly assign values to the attribute of the relation whose position is occupied by "variable" in the antecedent of rule. (If "variable" appears in multiple relations, the rule assigns values to the corresponding attributes of all of them.) As mentioned in the introduction, such attributes are called *intentional (virtual) attributes.*

The least fixpoint semantics are applicable in this case as well, although in a degenerate way. Since rules are assumed to be range-restricted, for every tuple in the relation where "variable" appears, the bindings in the antecedent of the rule determine the value of "expression." For equality (=) to be satisfied, the fixpoint semantics dictate that this is the value of the appropriate attribute of the given tuple in the relation. If a tuple in the relation where "variable" appears is associated with multiple bindings in the antecedent of the rule generating different values for "variable," then inconsistency arises and must be resolved.

2.2 Inconsistent sets of rules

Consider a set of derivation rules $\{r_i\}$. We say that $\{r_i\}$ is *inconsistent*, if there exists a database D such that for some fact α, one can derive both α and *not* (α) from D and $\{r_i\}$. We investigate the situations that produce inconsistencies. As long as no negative information is contained in the database, i.e., there is no tuple known not to exist in a relation, inconsistency cannot arise: only positive information is stored explicitly, and only positive information can be derived by Horn clauses. This may lead to the false conclusion that, by only defining Horn clauses as derivation rules, no inconsistency can arise. The following classical example by Stonebraker and Rowe (1986) shows two Horn clauses that are clearly inconsistent, thus proving that the above does not hold:

EMP(*name,title,type_of_desk*) $\rightarrow$ *type_of_desk = "steel"*,

EMP(*name, "chairman",type_of_desk*) $\rightarrow$ *type_of_desk = "wood"*.

The two rules offer two contradicting types for the desk of the chairman of the company. Our previous reasoning about when inconsistencies arise is still correct

though. The inconsistency arises because of the implicit negative information in the database that *not("steel"="wood")*. Negative facts on primitive relations are always implicitly part of the database, and inconsistencies are possible among derivation rules with such relations in their consequent. To the contrary, assuming that no explicit negative information is stored in the database,

no inconsistency can arise among derivation rules with user-defined relations in their consequent.

Equivalently, the two types of rules (with primitive and nonprimitive relations in their consequents) can be distinguished as follows. Let x and y be appropriately defined tuple variables, $q(D)$ be the set of tuples constructed from constants in D that have the same arity as $\langle x, y\rangle$ (the tuple formed by the concatenation of x and y) and satisfy some qualification q, $\mathbf{Q}$ be a user-defined relation, $x.att$ be the attribute *att* of x, and g be a function from the set of tuples with the same arity as $\langle x, y\rangle$ (domain) to the set of legal values for attribute *att* (range). A derivation rule with a user-defined relation in its consequent defines elements of that relation and has the general form

$$\langle x, y\rangle \in q(D) \rightarrow x \in \mathbf{Q}. \tag{4}$$

Since the database contains only positive facts, multiple rules that declare members of $\mathbf{Q}$ can never present a problem. To the contrary, a derivation rule with equality (=) in its consequent defines elements of a function i.e., values of the function on specific members of its domain, and has the general form[2]

$$\langle x, y\rangle \in q(D) \rightarrow x.att = g(x, y). \tag{5}$$

A function is constrained to return a unique value for every member of its domain. When multiple rules give values to $x.att$ in their consequents, they effectively define a function in a piecewise fashion. Inconsistency arises when a tuple X satisfies the antecedents of multiple rules, and the g functions used by the rules give different values on X. It can also arise within a single rule when a tuple X satisfies the rule's antecedent in association with multiple Y's, and the g function used by the rule gives different values for different Y's. For every such tuple X, the system must assign a value to $X.att$ based on those returned by all qualifying rules and for all qualifying Y tuples.

There are several criteria that can be used in determining attribute values when inconsistencies arise. In the next subsection, we give a brief overview of the criteria used by some existing systems and other related work. In the section after that, we develop a framework that subsumes all currently used criteria and also introduces some new ones that capture the rule semantics in cases where the known solutions fail.

2.3. Related work

To the best of our knowledge, the specific problem of resolving conflicts between rules deriving values for a virtual attribute has not been directly addressed in the context of deductive database systems. Similar problems, however, arise in systems supporting production rules, in handling inconsistent logic programs, and in handling multiple inheritance in generalization hierarchies.

Much of the work on resolving rule inconsistencies has been done in the context of expert systems that use *production rules*, i.e., condition-action pairs. In such systems, inconsistency arises when from a set of qualifying rules, precisely one has to fire. Then, conflict resolution of the form discussed in this paper is needed. Production rules can be used to imitate functional derivation rules: the action part of the rule can be an assignment of a value to a virtual attribute. Hence, conflict resolution is a common problem to both types of rules, and solutions to one of them are often applicable to the other one as well.

We are aware of two solutions currently in use for production rules, one that is best exemplified by the OPS5 system (Forgy, 1979) and another that is best exemplified by the Postgres system (Stonebraker, et al., 1988). OPS5 uses an elaborate criterion to resolve conflicts, which takes into account structural properties of the rules and other properties of the data involved. These include the complexity of the antecedents of the rules, the recency of the rules in conflict, and the recency of the tuples satisfying the rules. If the conflict cannot be resolved based on any of these properties, a rule is chosen arbitrarily, so that it is guaranteed that a single rule will fire.

Postgres, which is an extended relational database system supporting production rules, uses a very simple criterion at the expense of making the rulebase design more complicated. Each rule is assigned a priority. When rules are in conflict, the one with the highest priority is chosen. The main disadvantage of this method is that it puts the burden of assigning rule priorities to the rulebase designer. This approach is also taken by other recent extended relational database systems supporting production rules, such as Starburst (Widom, et al., 1991) and Ariel (Hanson, 1992). Further details on the techniques used in both OPS5 and Postgres are presented in Section 4.

The work on inconsistent logic programs is also related to the theme of this paper. The primary goal of such work is to present nontraditional logics that handle inconsistent beliefs. The essence of much work in that area is to define a multivalued logic and provide enhanced semantics so that resolution can be performed in such logics. Prominent examples of such investigations are those by Kifer and Lozinskii (1989) and Blair and Subrahmanian (1989). These offer insights on what the semantics of inconsistent programs should be, but are more applicable to logic programs where query answers can be disjunctive. For example, considering the enhanced, inconsistent, set of rules of Section 1, the answer to the question on the color of the shirt of Jane Blue could be "red *or* blue." In a traditional database context, which is the environment of interest in

this work, attribute values must be atomic, so the above approaches are not very helpful.

Finally, similar problems arise in resolving multiple inheritance in generalization hierarchies. Borgida addresses the issue in the form of exception handling, and defines semantics for a language that explicitly captures contradictions in an inheritance hierarchy (Borgida, 1985, 1988).

3. A general framework for resolving conflicts

3.1. Notation and resolution algorithm

For the remainder of this paper, we use the general form of (5) for functional derivation rules. Consider the following set of functional derivation rules assigning values to the same attribute of a tuple (relation):

$$\begin{aligned} r_1: &\quad \langle x, y_1 \rangle \in q_1(D) \rightarrow x.att = f_1(x, y_1), \\ r_2: &\quad \langle x, y_2 \rangle \in q_2(D) \rightarrow x.att = f_2(x, y_2), \\ &\vdots \\ r_m: &\quad \langle x, y_m \rangle \in q_m(D) \rightarrow x.att = f_m(x, y_m). \end{aligned}$$

In each rule, x is a tuple variable ranging over the relation whose virtual attribute is defined by the rule, whereas each y_i is a tuple variable representing the combination of the remaining nonprimitive relations that appear in q_i. Without loss of generality, we assume that the attributes of x and y_i that appear in the antecedent of r_i are not virtual, i.e., they are not defined by another set of rules. Otherwise, the relevant rules can be appropriately composed so that a rule set with the above property be obtained.

Every attribute of a relation can be thought of as a function from the tuples in the relation (domain) to the legal values of the attribute (range). Given a tuple as input, the function returns the value of the attribute as output. Assume that the function corresponding to attribute *att* is f. Let $SAT(X)$ be the set of rules generating values for *att* whose qualifications are satisfied by tuple X, i.e.,

$$SAT(X) = \{r_i \mid \exists Y \text{ such that } \langle X, Y \rangle \in q_i(D)\}.$$

Given a tuple $X, f(X)$ is calculated based on the values in $\{f_i(X,Y) \mid r_i \in SAT(X)\}$. This calculation is represented by a function u (u for unique), which is associated with the virtual attribute *att* and determines its value based on all the generated values. Specifically, given a tuple X and a query on $f(X)$ (that is on $X.att$), if conflict arises, the necessary information is given as input to u, which then computes the value of $f(X)$ and returns it as the answer to the query. In general, u is a user-defined function and depends on the semantics

of *att*. Section 5 contains some proposals on how u can be specified by the rulebase designer. The system can have a default function u, which is applied when nothing has been specified.

Based on the above formulation, u may be an arbitrary function. We expect, however, that in most cases, the semantics of the data will be such that it will be quite simple. In addition, the information considered by u may be dynamic or static, i.e., it may or may not depend on the current database state. In the former case, u may have to access the database or some statistics about the database kept in the catalogs. In the latter case, conflicts can be resolved during a preprocessing phase, so that no access to the data is needed at query time.

3.2. Classification of resolution criteria

Assigning a unique value to a virtual attribute using the values returned by all qualifying rules may be done according to many criteria. These can be broadly classified based on two aspects of them. First they are divided into *choice* criteria and *combination* criteria. When a choice criterion is applied, the value assigned to the virtual attribute is one of those returned by the qualifying rules. When a combination criterion is applied, the value assigned may be an arbitrary function that potentially takes into account all the values returned. Intuitively, the above distinction may be seen as one rule firing in choice criteria versus all rules firing in combination criteria.

Second, criteria are also divided into *value-based, rule-based*, and *mixed* criteria (where mixed criteria are combinations of the other two types). When a value-based criterion is applied, the value assigned to the virtual attribute is determined based on properties of the generated values themselves. When a rule-based criterion is applied, the value assigned is chosen based on properties of the rule that generates the value. Rule-based criteria are further divided into *user-specified* and *intrinsic* criteria. In the former, a rule is chosen based on information that the user associates with the rule. In the latter, a rule is chosen based on the form of the rule itself, particularly its antecedent and the set of tuples that satisfy it.

An interesting question that arises from the above is how the two classifications are related. Clearly, value-based criteria can be either choice or combination criteria. However, rule-based criteria can only be choice criteria. Since the rule alone determines the assigned value, that must be the value generated by the rule. All the above classes of criteria and their relationships are captured in an inheritance-style hierarchy in Figure 1.

In general, there exist applications requiring the use of any combination of the above types of conflict resolution criteria. Although the appropriateness of a criterion depends on the specific semantics of the virtual attribute, there are certain differences among the various categories mentioned above that are

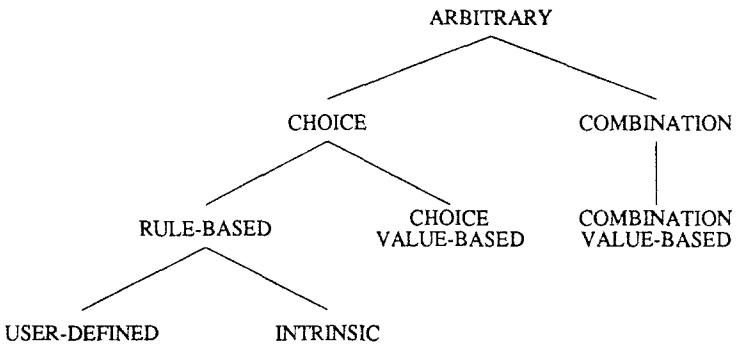

Figure 1. Classification of conflict resolution criteria.

helpful in identifying the type of criterion required. One such issue is difficulty of rulebase design. By definition, user-specified rule-based criteria leave much of the responsibility of resolving conflicts to the rule designer. Each rule must be associated with the appropriate information so that at conflict resolution time, the appropriate rule is chosen. When dealing with large rule-bases, this processes is quite sensitive and leaves the door open for errors to enter the system. In addition, when rules are composed, a special theory has to be developed for the calculation of the information that will be associated with the composite based on the information available for the base rules. As experience has shown, this is a nontrivial task, and most likely the results are ad-hoc (Shortliffe, 1976). Intrinsic and value-based criteria do not use any information supplied by the rule designer, and are therefore more robust and do not present the composition problem.

Another and more important issue is resolution granularity. In rule-based criteria, the smallest granule of resolution is the rule: rules are compared and one is chosen independent of the individual tuple X whose virtual attribute is being filled. In value-based criteria, the smallest granule of resolution is the tuple $\langle X, Y \rangle$: the result is based on the final values produced for the virtual attribute, which in turn depend on the specific tuples X and Y. Hence, value-based criteria provide a much finer distinction among conflicting values. Based on the above property, one can prove the following statement, which characterizes the cases where rule-based criteria can be used.

PROPOSITION. Consider a set of rules that defines the function $f(x)$, whose consequents correspond to the set of functions $\{f_i(x, y) \mid 1 \leq i \leq m\}$. Rule-based criteria may be used for the computation of $f(x)$ if and only if (a) there is a functional dependency $x \rightarrow y$ from x to y, or (b) for each rule r_i, either f_i is not a function of y or q_i can only be satisfied by a single instance of y.

Proof. Assume that there is a functional dependency $x \rightarrow y$. Then, for all $1 \leq i \leq m$ and any tuple X, there is a unique tuple Y such that $\langle X, Y \rangle$ satisfies the antecedent of r_i. This further implies that, for each tuple X, there is a unique value generated by r_i and the rule-based resolution is adequate. Similarly, assume that for each rule r_i, either f_i is not a function of y or q_i can only be satisfied by a single instance of y. In the former case, for each X, there is a unique value that f_i generates independent of Y. In the latter case, there is only one Y that can be used in association with any X, so again only one value is generated by r_i. Therefore, in all cases, rule-based resolution is again adequate. For the only-if part of the statement, one can follow the proof above in reverse order.

Based on the above discussion, we believe that value-based criteria are the most desirable. There are several examples of applications in the rest of the paper where the desirability of such criteria is demonstrated. Strangely enough, to the best of our knowledge, our work (Ioannidis and Sellis, 1989) has been the only study that considers resolving conflicts based on the candidate values for the virtual attribute. All other works have concentrated on various forms of rule-based criteria. Part of the reason may be that, in principle, value-based criteria require extensive interaction with the database at run time to resolve conflicts and are therefore computationally more expensive than rule-based. Specifically, value-based criteria require that all applicable rules are processed and then a value is computed, whereas rule-based criteria first choose a rule among those applicable and then process that rule alone. We do show, however, in Section 6.1 several cases of value-based criteria that can be managed at compile time, thus making them computationally attractive as well.

3.3. Resolution criteria

In this subsection, we examine several types of criteria, giving some insights into how u may operate in each case, together with some examples intuitively falling into the specific case. For ease of presentation, we always assume that precisely two rules are inconsistent. Generalizing the discussion to more than two conflicting rules is straightforward.

3.3.1. User-specified rule-based criteria. Such criteria are already being used by existing systems (Stonebraker, et al. 1988), and in our opinion, are the least desirable. The reason has been discussed in Section 2.3: in addition to specifying a function u for every virtual attribute, the rulebase designer has to deal with properties of each rule individually, which with few exceptions would otherwise be unnecessary. Moreover, this criterion requires a good knowledge of the results of the rules in advance, at rule design time, so that their properties can be specified accordingly.

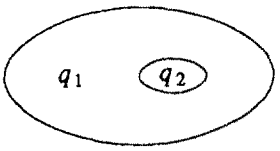

Figure 2. Qualification inclusion.

The most straightforward example is assigning priorities to rules, which is the hard-wired conflict resolution scheme in Postgres (Stonebraker, et al. 1988). Among the qualifying rules, the one with the highest priority is chosen by u. If x is the tuple variable ranging over the relation of the virtual attribute, then the rule of choice r_j must satisfy

$$r_j \in SAT(x) \textbf{ and } priority(r_j) = \max_i(\{priority(r_i) \mid r_i \in SAT(x)\}). \quad (6)$$

Another rule property that might sometimes be useful in resolving conflicts is certainty factor. Several expert systems, e.g., MYCIN (Shortliffe, 1976), do not trust all rules alike and assign certainty factor to them. A reasonable way to resolve conflicts is to use the value returned by the most trustworthy rule. Systems that do assign certainty factors to rules and want to use the one with the highest certainty factor in case of conflict, essentially use formula (6) as their choice criterion with certainty factor replacing priority.

3.3.2. Intrinsic rule-based criteria. In this case, rules are distinguished based on their antecedents. Function u takes as input the sets of tuples that qualify under each rule, and based on some properties of the sets, chooses to apply one of the rules. Some intuitive functions that could play the role of u are discussed below.

Antecedent inclusion. Suppose that for all databases $D, q_2(D) \subseteq q_1(D)$, i.e., the antecedent of r_2 is strictly less general than that of r_1. This inclusion holds at the expression level, and it does not depend on the database contents. We believe that the intuition behind the two rules is that the more general one applies only when the less general one does not. Schematically, r_1 actually applies in the region between the borders of q_2 and q_1, i.e., in the region of $q_1 - q_2$ (Figure 2).

One could define the two rules in a nonconflicting way by using "$q_1 \wedge not(q_2)$" as the antecedent of r_1. In the case of a chain of rules, each one of which is less general than the other, the size of the antecedent of each rule increases by a factor of 2. In addition to the task of the rule designer becoming more tedious, the possibility of errors increases as well. For all these reasons, it is preferable to define the rules in a conflicting way and resolve the conflicts at a higher level. Using the x tuple variable again, the least general rule r_j is identified based on the condition

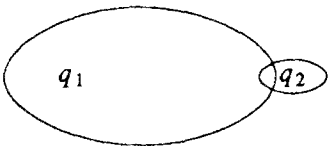

Figure 3. Size minimization.

$$q_j(D) = \min_i(\{q_i(D) \mid r_i \in SAT(x)\}), \tag{7}$$

where *min* is with respect to $\subseteq$ in (7).

Alternatively, one can approach this criterion by interpreting the less general rules as exceptions to the more general ones. Normally, f_1 is used, with an exception when q_2 holds, in which case f_2 is used. As an example, consider a rule that specifies that "all professors teach 3 semester courses per year" and another one that specifies that "the chairman teaches 1 course per year." Intuitively, the second rule is interpreted as an implicit exception to the first one, so that the chairman, who is also a professor, only teaches one course. This exception mechanism is captured by f if it is defined by (7). In general, the antecedent inclusion criterion is not always applicable, since conflicts may arise between rules whose antecedents are incommensurate. Whenever applicable, however, it seems to be the right choice.

Size minimization. A similar criterion, based on the same intuition about exceptions, uses the sizes of the qualifying sets of the two rules. The less restrictive rule, i.e., the one satisfied by more tuples in the database, applies only when the more restrictive one does not. As in the antecedent inclusion case, schematically, r_1 applies in the region of $q_1 - q_2$ (Figure 3).

Again the main reason to define rules so that they conflict is convenience. The more restrictive rule is interpreted as an exception to the less restrictive one, and it is used whenever it is applicable. If r_j is the rule of choice, then it satisfies

$$r_j \in SAT(x) \textbf{ and } size(q_j(D)) = \min_i(\{size(q_i(D)) \mid r_i \in SAT(x)\}), \tag{8}$$

where *min* is with respect to integer inequality $\leq$ in (8).

As an example, taken from the Greek military conscription law, consider a rule that specifies that "all married males with one child serve in the army for 1 year" and another one that specifies that "all males who are Jehovah's witnesses serve for 4 years" (without ever carrying a gun). There are many more people with one child than there are Jehovah's witnesses in Greece. Intuitively, the second rule is interpreted as an implicit exception to the first one, so that a Jehovah's witness that has one child is required to serve for four years. This exception mechanism is captured by f if it is defined by (8).

Clearly, if antecedent inclusion is satisfied, then size minimization is as well. The former is a stronger requirement than the latter. In some sense, size minimization is more desirable, since it is more likely to produce a resolution to the conflict, and simultaneously less desirable, since it may fail to capture the intuition behind the rules, if the two sizes are about the same. This may be avoided if one requires that there is a substantial difference between the two sizes, by defining u appropriately.

3.3.3. Choice value-based criteria. In current systems, conflicts are never resolved according to the values given to the virtual attribute by the rules. There are several applications, however, where the values returned by the rules are the decisive factor on which rule to apply. For example, optimization problems can be formulated as a function f (the value of the virtual attribute) being optimized according to some criterion, e.g., minimized, maximized. The rule to be used should be the one that achieves the optimal value for f. No other resolution scheme can achieve the same semantics.

As a concrete example, consider a car dealership that uses a rule that specifies that "all cars under 1 year old are 100% warranted" and another rule the specifies that "all cars that have been driven more than 5000 miles are 50% warranted." For a car that is less than one year old but has been driven 8000 miles, the dealer uses clause like "whichever comes first," which can be translated to "whichever produces the least coverage." That is, conflicts are resolved based on the value of coverage percentage, and in particular by using the rule that provides the least coverage. For this example, u is *min*, and the criterion can be represented as

$$coverage(x) = \min_i(\{coverage_i(x) \mid r_i \in SAT(x)\}),$$

where again x is the tuple variable ranging over the relation with the coverage attribute. Note, that a customer-chosen conflict resolution scheme would choose the rule providing the maximum coverage! This further supports our claim that only the semantics of the virtual attribute determines the correct conflict resolution scheme.

3.3.4. Combination value-based criteria. There are several applications whose semantics require that the value of a virtual attribute is not chosen from the values generated by the qualifying rules, but that it is determined by some computation that takes into account all these values. Such cases can be handled by combination criteria, which allow arbitrary computations for the virtual attribute value. As mentioned above, such criteria are always value-based or mixed.

As a concrete example, consider a distributed consensus problem, where many different sources offer a value for a certain parameter. This process can be formulated with several rules, each representing one of the different sources, assigning a value to an attribute representing the parameter. The final value of the attribute may be determined from the proposed values according to some

arbitrary criterion, e.g., taking the average of all values except the greatest and the smallest.

3.4. Comments

The moral one can draw from the above discussion is that there is no universal criterion that can be assumed as the default, which systems can apply without any user-provided knowledge about the semantics of the data. Moreover, the various types of criteria specified are independent of each other. For example, a system that assigns priorities to rules and uses (6) to resolve conflicts cannot always capture the semantics of resolving conflicts by choosing the rule that produces the minimum value. In the car dealership example used in Section 3.3.3, one may argue that if one assigns higher priority to the rule that provides only 50% coverage than to the one that provides 100% coverage, conflicts are resolved by choosing the first rule as desired. If, however, the rules were of the form "all cars lose 10% of their warranty per year as they age" and "all cars lose 10% of their warranty per 5000 miles they make," it would be impossible to achieve the same conflict resolution by using priorities, simply because the rule results depend on the specific values of age and mileage of the car.

4. Case studies

In this section, we present several inconsistent rule sets and show how they are handled by previous proposals as well as our approach.

Example 1 (Stonebraker, et al. 1988). Suppose that a relation **EMP**(*name,age,sal*) is given, where *sal* is an attribute whose value may be stored explicitly or may be virtually defined. Assume that the following two rules, r_1 and r_2, derive Mike's salary:

$$r_1 : \mathbf{EMP}(name,age,sal) \wedge \mathbf{EMP}(name1,age1,sal1) \wedge name = \text{"}Mike\text{"} \wedge name1 = \text{"}Bill\text{"} \rightarrow sal = sal1,$$

$$r_2 : \mathbf{EMP}(name,age,sal) \wedge \mathbf{EMP}(name1,age1,sal1) \wedge name = \text{"}Mike\text{"} \wedge name1 = \text{"}Fred\text{"} \rightarrow sal = sal1.$$

We look first at the solution Postgres offers for these two conflicting rules. Rules are associated with priorities, and in the case of a conflict, the rule with the highest priority is chosen to fire. To avoid inconsistencies, r_1 is given a priority of 5, while r_2 is given a priority of 7 (Stonebraker, et al. 1988). Asking for Mike's salary returns Fred's salary.

Under our framework, both rules represent distinct points in the **EMP** $\times$ **EMP** data space, and any intrinsic conflict resolution scheme is useless. Assuming

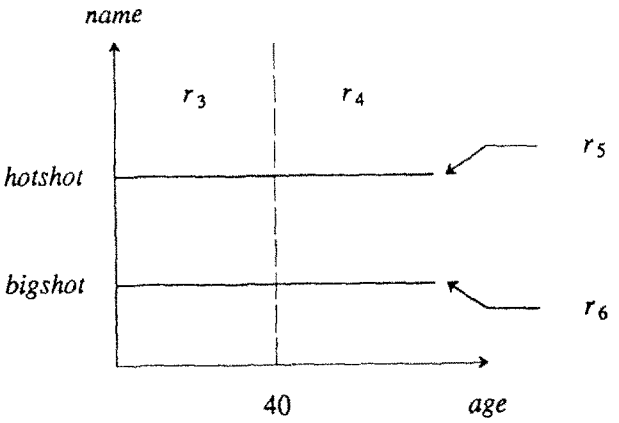

Figure 4. Qualifications of rules r_3–r_6.

also that the salaries produced by the rules do not affect the decision on the final value, a user-defined rule-based criterion based on priorities is reasonable. Hence, if x ranges over **EMP**, $x.sal$ is set by the rule r_j that satisfies the following:

$$r_j \in SAT(x) \textbf{ and } priority(r_j) = \max_i(\{priority(r_i) \mid r_i \in SAT(x)\}).$$

The effect of such an assignment is exactly the same as with the Postgres rules.

Example 2 (Stonebraker and Rowe, 1986). Suppose that a relation **EMP**(*name, age,salary,desk*) is given, where *desk* is a virtual attribute defined as follows:

r_3 : **EMP**(*name,age,salary,desk*) $\wedge$ *age*<40 $\rightarrow$ *desk* = "*steel.*"
r_4 : **EMP**(*name,age,salary,desk*) $\wedge$ *age*$\geq$40 $\rightarrow$ *desk* = "*wood.*"
r_5 : **EMP**(*name,age,salary,desk*) $\wedge$ *name* = "*hotshot*" $\rightarrow$ *desk* = "*wood.*"
r_6 : **EMP**(*name,age,salary,desk*) $\wedge$ *name* = "*bigshot*" $\rightarrow$ *desk* = "*steel.*"

Assuming that "hotshot" is 32 years old, there is an inconsistency in the above rules (r_5 and r_3).

For this example, Stonebraker et al. (1988) suggest to give priority 1 to r_3 and r_4, and priority 2 to r_5 and r_6 (Stonebraker and Rowe, 1986). Hence, "hotshot" is assigned a wood desk, since r_5 has higher priority than r_3.

It is clear, however, that the information about "hotshot" corresponds to a fact known to the rule designer. In the **EMP** data space, the qualifications of the above rules are shown in Figure 4 (for simplicity, we only show the *age* and *name* coordinates).

One can see that rule r_5 corresponds to a line while rule r_3 defines a much larger area. Hence, use of priorities is unnecessary; the semantics of the rules

imply that conflicts can be resolved by size minimization of the qualifying sets of tuples using formula (8).

Example 1 shows that the priority-based resolution scheme of Postgres can be modeled under our framework, while Example 2 identifies cases where an intrinsic rule-based criterion is much more appropriate than priorities. Next, we present a more complicated example, where a combination of various conflict resolution schemes is used.

Example 3 (Forgy, 1979). OPS5 is a production system and uses rules of the following form:

$$x \in q(D) \rightarrow action(x).$$

Function *action* depends on the qualifying tuple that binds to x, and it can be an insertion, deletion, modification, or execution of a general procedure. Tuple insertions, deletions, and modifications make rules to fire (by satisfying their antecedent), which in turn can create a cascade of rules becoming applicable to fire. The problem of conflict resolution arises in OPS5 when multiple rules are applicable to fire. The set of rules that is applicable to fire at any one time is called the *conflict set*; a rule is in the conflict set, if there is some x in the database that makes the antecedent of the rule true. Note that the problem faced by OPS5 is not precisely one of choosing among multiple values for a virtual attribute, which is the main problem addressed in this paper. Solutions to the former problem, however, could very well serve as solutions to the latter. Thus, it is important to show that those solutions can be expressed in the general framework of Section 3 as well.

An abstraction of the conflict resolution scheme used in OPS5 uses the following criteria in the order they are given below. The exact algorithm is not presented here because it makes use of details of the system that are of no interest to this paper (Forgy, 1979).

1. choose the rule with the most recent tuples in $q(D)$; if several are equally recent
2. choose the rule with the highest number of literals; if several have the same
3. choose the rule with the highest number of constants; if several have the same
4. choose the rule introduced in the conflict set most recently; if several are equally recent
5. choose an arbitrary rule.

We see that OPS5 uses a mix of time-based and size-based criteria. Rules 1 and 4 depend on the time the tuples were produced and the time the rules were put in the conflict set respectively. Rules 2 and 3 compare syntactic characteristics of the rules to indirectly capture differences in the sizes of the qualifying sets.

Maximizing the number of literals or constants is an approximation of minimizing the size of the sets $q_i(D)$.

Assume that tuples are assigned timestamps when they are inserted into the database, while rules are assigned timestamps when they are put in the conflict set. Let *time* be the function that returns the timestamp associated with its input. The above criteria can then be modeled in our general framework as follows (for brevity we give only the criterion function, not the whole u function):

1. $max(time(q_i(D)))$
2 and 3. $min(size(q_i(D)))$
4. $max(time(r_i))$
5. $random(r_i)$

Clearly, despite the complexity of the criteria used in OPS5, they can be easily represented in the general framework that we have suggested, thus showing the power of the mechanism.

Example 4 (Kung, et al., 1986). Last we present an example where a value-based criterion is used for conflict resolution. Some implementations of heuristic search of large graphs stored in a database have to rely on nonfunctional updates of virtual attributes (Kung, et al., 1986). Suppose that a map is recorded in the form of a directed graph in a relation **MAP**(*source,dest,cost*). Attribute *cost* represents the cost of the arc (*source* $\rightarrow$ *dest*). The problem is to find the least expensive path between two nodes "start" and "finish." This is done with the help of a relation **STATES**(*dest,cost*), which records the least cost of going from "start" to every node *dest* in the graph. The algorithm repeatedly updates the cost of reaching a node *dest* with the cost of reaching a neighbor node *sdest* plus the cost of going from *sdest* to *dest*. This is achieved by repeatedly using the single rule.

$$r: \textbf{STATES}(sdest,scost) \wedge \textbf{MAP}(msource,mdest,mcost) \wedge \textbf{STATES}(dest,cost)$$
$$\wedge sdest = msource \wedge mdest = dest \rightarrow cost = scost + mcost.$$

Clearly, the basic step of the algorithm introduces nonfunctional updates. If the node *dest* can be reached through multiple paths, rule r tries to assign conflicting values to the cost of this node. The semantics of the update are such that the minimum cost is chosen (Kung, et al., 1986). This semantics cannot be achieved by any rule-based criterion, since all cost derivations are generated by a single rule. Instead, a value-based criterion must be used, which is simply expressed by associating with the virtual attribute cost of **STATES** the function

$$f(x) = \min_i(\{f_i(x,y)\}),$$

where each $f_i(x,y)$ is in our case the value *mcost*+*scost* generated from one of the various *sdest* nodes. The above method can be actually used in any kind of

search algorithm, such as A* and Branch & Bound, since it allows the user to specify in a rigorous way what the choice criterion is.

5. Specifying conflict resolution criteria

In order for the above conflict resolution framework to be implemented in a system, a language must be developed in which resolution criteria will be expressed. Without proposing the syntax of a full language, in this section, we focus on the problem of identifying the parts of a database with which such criteria should be associated. In general, conflict resolution information can be specified as a property of (a) the database schema or (b) the rules themselves. In the former case, a resolution criterion is associated with the virtual attribute in its schema definition, while in the latter case, it is associated with the rules that try to assign values to it.

Note that the process of conflict resolution, as described in this paper, is very similar to the process of *inheriting* values in object-oriented systems, and more generally, handling generalization hierarchies. The two techniques mentioned above, and detailed below, are related to the concept of *static* and *dynamic* inheritance, as they try to resolve on conflicting values based on some static (schema) or dynamic (rule modules or programs) information respectively.

5.1. Specifying resolution criteria in association with the schema

The first solution comes up naturally since we are limiting ourselves to rules assigning values to virtual attributes; these attributes are specified in the schema of the relations that contain them. As with integrity constraints, one can specify in the schema the way conflicting values are to be used in order to derive a single value for an attribute. For example, given the relation

EMP(*name,age,sal,years_employed*),

where *sal* is a virtual attribute, one can specify together with the type of the attribute, one or more of the following

resolve on {
rule **based on** *min* **of** *antecedent size*;
value **based on** *min*}.

The meaning of the first line is that rule-based resolution should be employed, choosing the qualifying rule that has the fewest tuples satisfying its antecedent. The second line indicates that value-based resolution should be employed, picking the minimum among all generated values (i.e., salary values). The order in which the above criteria are specified indictates the order in which they should

be applied. In the above example, if all applicable rules have qualifying sets of the same size and the first criterion is ineffective, resolution should be made by picking the minimum value.

A significant advantage of specifying resolution information with the schema, is that rule compilation can take advantage of this information to produce efficient code, in the same sense that integrity constraints can be used when compiling programs to avoid illegal updates. In the case where rules are used in programs to assign values to virtual attributes of relations (as has been the case with all our examples), these programs can be compiled taking into account the information available about conflict resolution. The idea is that all rules relevant to setting the value of a particular attribute will be examined, and a "case" statement will be built. Such a statement will examine the qualifications of all the rules that may be applicable and among them enforce the conflict resolution criterion to pick a single value (rule-based resolution), or combine the values (value-based resolution). This certainly results to more efficient code, compared to compiling each rule individually, since the conflict resolution code is included in the program code and does not need to be applied every time a conflict arises. A question of interest that also arises is how efficiently qualifications of rules can be checked; we address this issue in the next section.

5.2. *Specifying resolution criteria in association with the rules*

The above way of specifying conflict resolution strategies is useful when all applications written on the same database are to follow the same criteria for selecting among conflicting values. This, however, may not always be true. For example, consider two applications that use two different sets of rules to assign salaries to employees. The semantics of one of them may require that the minimum value among those generated by all qualifying rules should be assigned each time, whereas the semantics of the other may require that the maximum value should be assigned. Clearly, in such a case, different resolution criteria must be associated with each rule set; associating them with the schema is inappropriate.

We assume that rules are organized into *rule modules*. Modules define sections of a rule program that contain rules referring to the same entities, or attempt to solve the same problem. This implies that only rules within one module should be considered when conflict resolution takes place. Hence, a user may limit the scope of a conflict resolution criterion to a particular rule module that is appropriate for a specific application. Obviously, by putting all rules into a single module we have the same effect as having no modules at all. Deciding the granularity of rule modules clearly depends on the application.

Given a rule module, resolution criteria may be declared and associated with the module as in the previous subsection. In addition to what was given earlier, the user has to specify which attribute is to be resolved using these criteria. As

an example, the following two modules are associated with different criteria to resolve conflicts.

RULEMODULE *RM1*

r_1 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $15 \leq years\_employed \leq 40 \rightarrow$
$sal = 3K * age + 20K.$
r_2 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $10 \leq years\_employed \leq 25 \rightarrow$
$sal = 4K * age - 20K.$
r_3 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $5 \leq years\_employed \leq 20 \rightarrow$
$sal = 5K * age - 40K.$

resolve *sal* **on**
rule **based on** *min* **of** *antecedent size*;
rule **based on** *max* **of** *priority*;
value **based on** *min*}

endmodule

RULEMODULE *RM2*

r_4 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $20 \leq age \leq 30 \rightarrow$
$sal = 3K * years\_employed + 20K.$
r_5 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $25 \leq age \leq 40 \rightarrow$
$sal = 4K * years\_employed - 20K.$
r_6 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $35 \leq age \leq 50 \rightarrow$
$sal = 3.5K * years\_employed - 25K.$

resolve *sal* **on** {
rule **based on** *min* **of** *antecedent size*;
value **based on** *avg*}

endmodule

When a program uses one of these modules, conflict resolution will be made according to the criteria associated with the module used. In addition, the same compilation techniques with the ones described at the end of the previous subsection apply here as well.

In summary, conflict resolution criteria can be associated either with the database schema or with rule modules, depending on the semantics of the database and its applications. The design of languages for specifying resolution criteria and the level of detail in which such specifications need to be made are interesting issues of future research.

6. Rule indexing for efficient conflict resolution

In this section, we address the issue of efficiency in identifying the applicable rules producing values for a virtual attribute of a given tuple. Moreover, we address the issue of efficiency in resolving conflicts among multiple such rules. Given a large set of rules and a query asking for the value of a virtual attribute, identifying the relevant rules is time consuming. Several proposals have been suggested that speed up this process; some are based on putting special types of locks on the relevant attributes, tuples, and/or relations and using the locking mechanism to avoid looking at irrelevant rules (Stonebraker, et al., 1988); others are based on indexing the antecedents of rules using multidimensional data structures (Stonebraker, et al., 1987; Sellis and Lin, 1992). For the discussion in this section, we assume that some form of multidimensional index is used to index the antecedents of the rules, like an R^+-tree (Sellis, et al., 1987).

6.1. Application of geometric indexing of rule regions to conflict resolution

Consider a set of rules assigning values to the same virtual attribute. The antecedents of the rules define a multidimensional space whose dimensions are the various relation attributes that participate in some restriction in at least one of the antecedents. In this space, each antecedent specifies a region, within which the corresponding rule is applicable, like in Figures 2 and 3. Conflicts arise when two such regions from different rules overlap. If the space is indexed with an R^+-tree, it is divided into nonoverlapping regions which are in one-to-one correspondence with the leaves of the index. Each leaf contains pointers to the rules that are applicable in the corresponding region. When at run time the value of the virtual attribute defined by the rules is requested for some tuple, the index is probed, the set of rules satisfied by the tuple is identified, and conflict resolution is applied on that set. For rule-based resolution criteria (both user-defined and intrinsic), this will consist of identifying one rule from that set and processing it. For value-based criteria, this will consist of processing all the rules in the set and then identifying one value from the set of generated ones. Clearly, it is desirable that the conflict resolution process is done at compile time and that its result is incorporated into the index. Especially for value-based criteria, which first process all applicable rules and then make a final choice for the attribute value, tremendous savings can be realized if the index takes into account the conflict resolution scheme, so that only one rule is processed.

Below, we attempt to characterize the conflict-resolution criteria that can indeed be incorporated into an index so that at run time only the rule that will generate the desired attribute value is processed. We examine each of the two types of criteria (rule- and value-based) and discuss the run-time performance savings in each case. In what follows, we assume that there are R rules defining values for the attribute concerned and G regions in the space determined by the

antecedents such that in each region a different non-empty subset of the rules is applicable. The above two parameters are related with the following inequality: $G \leq 2^R - 1$.

6.1.1. Rule-based criteria. This is a rather straightforward case, because the complete information used for resolution at run time is available at compile time as well. Moreover, no additional dimensions need to be added to the index. For each region formed in the index, using the relevant rule properties, the resolution criterion is applied on all the rules that qualify in the region. The index then associates with that region a pointer to the rule chosen by the above process alone instead of associating with it pointers to all qualifying rules. At run time, the index will lead directly to the appropriate rule for processing.

The above holds for both user-defined and intrinsic criteria. The only difference between the two cases is that the resolution criterion itself may be more complicated and time consuming in intrinsic criteria than in user-defined ones. For example, in the size minimization (intrinsic) criterion, after the sizes are obtained, the process becomes equivalent to resolution based on user-defined rule priorities (where size plays the role of inverse priority). However, obtaining these sizes requires that the rules be processed against the database first, an operation that can be arbitrarily costly. Even if only estimates of the sizes are obtained by using some database statistics, the overall cost will probably be significantly higher. For the priority-based criterion, incorporating the resolution strategy into the index will require $O(GR)$ time, whereas if resolution was left for run time, it would cost $O(R)$ time for each probe. Hence, for heavily accessed attributes, resolving at compile time based on rule priorities is beneficial, while for lightly accessed attributes it is not. A similar trade-off exists for the size minimization criterion as well.

The antecedent inclusion (intrinsic) criterion is rather interesting with respect to its cost. In general, antecedent inclusion is an NP-Complete problem (Chandra and Merlin, 1977; Aho, et al., 1979), which would imply that the whole criterion is rather costly. However, in order for this criterion to be adequate for resolving conflicts between any pair of the available rules, the entire set of rules must form a linear chain with respect to antecedent inclusion. In this case, there is only a linear number of regions formed in the index, and resolution for all of them can be achieved by sorting the rules based on their antecedents. Assuming an exponential algorithm for antecedent inclusion, that would require $O(2^L R \log R)$ time, where L is some metric of the size of the rules. Observe that the cost per probe for run-time resolution would be $O(2^L R)$. Hence, in this case, it is almost always beneficial to incorporate the resolution in the index at compile time.

6.1.2. Value-based criteria. This is a much more complex case than the previous one, which on the other hand offers the opportunity for significantly higher savings when certain conditions hold. The main difference from the previous case that introduces the added complexity is that the information used for resolution at

run time is not readily available at compile time. That is, the candidate attribute values generated by the rules are unknown. Moreover, there may be many values generated by the same rule for a given tuple. Not only do the above characteristics increase the time complexity of resolving conflicts at compile time and incorporating the result into the index, but they also make such resolution quite often impossible. Precisely characterizing when this is possible is a rather hard problem on which no formal results have been obtained. The following discussion provides some insight into the problem and we believe points to some quite useful special cases.

With few exceptions, compile-time value-based resolution is possible only with choice criteria. Otherwise, if u returns the result of some manipulation of several of the generated values, then by definition, it is not one rule that should be processed at run time, but many. The exceptions are quite rare and represent cases where a *new* rule can be constructed that replaces the original ones, e.g., if u returns the average value in its input set and all rules generated a single value for each tuple. In the rest of the discussion, we only consider choice criteria.

A major difference between value-based and rule-based resolution is that, in the former, there is no unique rule that is chosen for each of the regions in the index defined by the antecedents of the rules. The rule of choice depends on the input parameters of the f_i functions, and these parameters should be represented as additional dimensions in the index (if they are not already there). Thus, compile-time resolution in this case consists of identifying regions in the expanded space where a unique rule can be chosen to be processed at run time. Abstractly, this process will be done separately for each of the original regions of the index and will involve identifying nonoverlapping regions in the corresponding subspace of the newly introduced dimensions. Since function u and the f_i functions may have arbitrary forms, this region identification (and therefore for whole concept of value-based, compile-time resolution) is not always possible.

Despite the above, there are many practical cases where u and the f_i functions are of simple enough form that compile-time resolution is achievable. In the remainder of our discussion, we concentrate on a very common case, where the virtual attribute is numeric and u returns a value based on the total order of the real numbers, e.g., u is *min, max,* or *median.* Even for this small (but important) subclass of functions, there is no guarantee that the region identification mentioned above can be done. For example, with u =*min*, the problem is reduced to solving the equations that equate each pair of the f_i functions and then using the roots as the region borders determining where each function generates indeed the minimum. This is not doable, however, if these equations cannot be solved, e.g., if some of the f_i functions are polynomials of degree higher than 4. Our belief is that, in most applications, the f_i functions will not be very complex, and that analytical ways will be available for region identification. Below, we concentrate on one such example, where all the f_i functions are linear and have a single common input attribute.

For the special case mentioned above, we investigate the time complexity of value-based, compile-time resolution. Because of the simple form of the f_i functions, the number of regions identified by pairwise comparisons of the functions (intervals on the real line) is at most $R(R-1)/2$. These regions will be merged with the original regions in the index determined by the rule antecedents. Therefore, the complexity of the process is $O(GR^2)$. When all rules are applicable in the entire space defined in the original index (i.e., when no rule has a nontrivial antecedent), the choice function is *min* or *max*, and the f_i functions are linear and have a single common input attribute, compile-time resolution is quite efficient. Based on the algorithm and analysis above, the overall complexity would be assumed to be $O(R^2)$ since $G = 1$. However, one can show that there are at most R regions that are of interest, which can in fact be identified in *expected* time that is linear in the number of rules, so that the overall expected complexity is brought down to $O(R)$.

Hence, we see that value-based resolution is not always possible, and whenever it is, it is more expensive than the other type of resolutions. However, the savings realized by not having to process multiple rules at run time for each probe to the virtual attribute, should more than offset the time spent in setting up the index.

6.2. *An example*

To illustrate the process of compile-time conflict resolution, we present a simple example where conflicts are resolved by choosing the minimum value among those generated by the conflicting rules. We chose a value-based criterion, because as mentioned above these are the most complex. Consider the following three rules, which are similar to those used in Section 5.1:

r_1 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $30 \leq years\_employed \leq 60 \rightarrow$
$sal = 3K * age + 20K.$

r_2 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $20 \leq years\_employed \leq 50 \rightarrow$
$sal = 4K * age - 20K.$

r_3 : **EMP**(*name,age,sal,years_employed*) $\wedge$ $0 \leq years\_employed \leq 40 \rightarrow$
$sal = 5K * age - 40K.$

The antecedents define a 1-dimensional space (on *years_employed*), and the regions defined by them are shown in Figure 5. The three functions in the consequents of the rules are plotted in Figure 6. The values of *age* where these functions cross over each other are as follows: for f_1, f_2 it is $age = 40$; for f_1, f_3 it is $age = 30$; and for f_2, f_3 it is $age = 20$. Adding *age* as a dimension in the space of Figure 5, and taking into account the crossover points of the three functions, we have

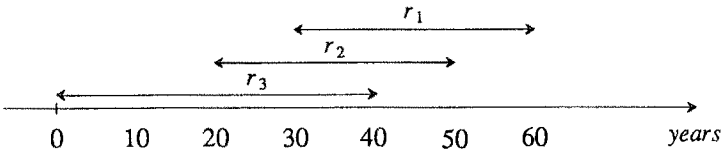

Figure 5. Regions of rules before conflict resolution.

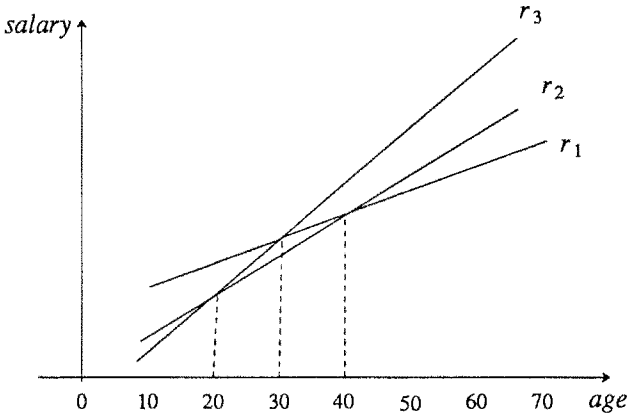

Figure 6. Plots of functions.

three new regions, which are now nonoverlapping. This is shown in Figure 7.

Given any tuple at run time, i.e., any specific values of *years_employed* and *age*, there is a unique region that it falls in, i.e., there is a unique rule associated with it. This rule can be identified very fast by using an index on the nonoverlapping regions of Figure 7. This results in avoiding any extra work associated with trying all applicable rules and applying the conflict resolution scheme.

7. Conclusions

In this paper, the problem of modeling conflict resolution schemes for inconsistent sets of virtual attribute rules has been addressed. We have presented a general framework that captures all previously suggested solutions as well as some new ones that we have proposed, thus proving its usefulness in a database context. We have also argued that simple solutions, such as hard-wired priorities, are not always useful, and that often user-defined schemes must be specified for successful conflict resolution according to the virtual attribute semantics. Finally, we have studied the use of multidimensional indices in performing conflict resolution

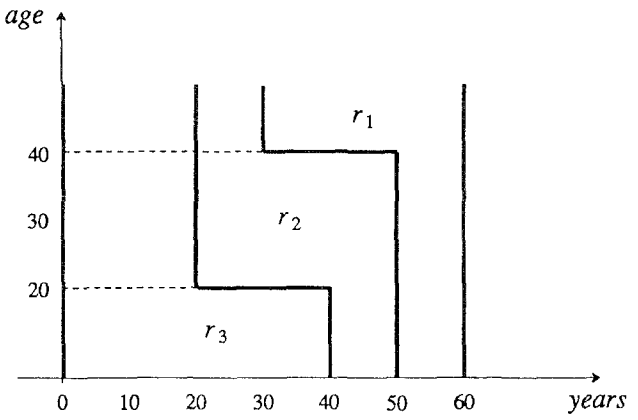

Figure 7. Regions of rules after conflict resolution.

at compile time, which when applicable may have significant implications on run-time performance.

We believe that more work is needed in the direction of obtaining a better understanding of the conflict resolution problem and devising better techniques for it. As future interesting problems we view the following.

1. The comprehensive study of storage structures that speed up the process of conflict resolution, based on the discussion of Section 6. The work of Sellis and Lin (1992) is a first step in this direction.
2. The investigation of implementation techniques for user-defined conflict resolution schemes, specified on each virtual attribute at schema definition time and triggered when needed.
3. The application of these ideas to the general conflict resolution problem. For example, conflicting integrity constraints must not exist in the system. Based on our framework, one could possibly define conflicting integrity constraints, allowing only one of them to be enforced in case of conflicts.

Notes

1. All relations appear in bold.
2. We employ the convention that the virtual attribute is always on the left-hand side of = in the consequent of functional derivation rules.

References

Aho, A., Sagiv, Y., and Ullman, J. (1979). Equivalences among Relations Expressions. *SIAM Journal on Computing, 8*, 218–246.

Aho, A. and Ullman, J. (1979). Universality of data retrieval languages. *Proc. 6th ACM Symp. Principles of Programming Languages* (pp. 110–117). San Antonio, TX.

Blair, H.A. and Subrahmanian, V.S. (1989) Paraconsistent Logic Programming. *Theoretical Computer Science, 68*, 135–154.

Borgida, A. (1985). Language Features for Flexible Handling of Exceptions in Information Systems. *ACM Transactions on Database Systems, 10*, 107–131.

Borgida, A. (1988). Modeling class hierarchies with contradictions. *Proc. 1988 ACM-SIGMOD Conf. Management of Data* (pp. 434–443). Chicago, IL.

Chandra, A.K. and Merlin, P.M. (1977). Optimal implementation of conjunctive queries in relational data bases. *Proc. 9th Ann. ACM Symp. Theory of Computing* (pp. 77–90). Boulder, CO.

Forgy, C.L. (1979). On the Efficient Implementation of Production Systems. Ph.D. dissertation, Carnegie-Mellon University, Department of Computer Science.

Hanson, E. (1992). Rule testing and execution in Ariel. *Proc. 1992 ACM-SIGMOD Conf. Management of Data* (pp. 49-58). San Diego, CA.

Ioannidis, Y. and Sellis, T. (1989). Conflict resolution of rules assigning values to virtual attributes. *Proc. 1989 ACM-SIGMOD Conf. Management of Data* (pp. 205–214). Portland, OR.

Kifer, M. and Lozinskii, E. (1989). RI: A logic for reasoning with inconsistency. *Proc. 4th Symp. Logic in Computer Science* (pp. 253–262). Cambridge, MA.

Kung, R-M., Hanson, E., Ioannidis, Y., Sellis, T., Shapiro, L., and Stonebraker, M. (1986). Heuristic Search in Data Base Systems. In L. Kerschberg (Ed.), *Expert Database Systems, Proceedings from the First International Workshop.* (pp. 537–548). Menlo Park, CA: Benjamin/Cummings.

Nicolas, J.M. and Gallaire, H. (1978). Data Base: Theory vs. Interpretation. In H.Gallaire and J. Minker (Eds.), *Logic and Data Bases* (pp. 33–54). New York: Plenum Press.

Sellis, T. and Lin, C-C. (1992). A Geometric Approach to Indexing Large Rule Bases. In A Pirotte, C. Delobel, and G. Gottlob (Eds.), *Advances in Database Technology – EDBT '92*, (pp. 405–420). Berlin: Springer-Verlag.

Sellis, T., Roussopoulos, N., and Faloutsos, C. (1987). The R^+-tree: A dynamic index for multi-dimensional objects. *Proc. 13th Int. VLDB Conf.* (pp. 507–518). Brighton, England.

Shortliffe, E.H. (1976). *Computer-Based Medical Consultations*: MYCIN. New York: Elsevier.

Stonebraker, M., Hanson, E., and Potamianos, S. (1988). The Postgres Rule Manager. *IEEE Transactions on Software Engineering, 14*, 897–907.

Stonebraker, M. and Rowe, L. (1986). The design of Postgres. *Proc. 1986 ACM-SIGMOD Conf. Management of Data*, (pp. 340–355). Washington, DC.

Stonebraker, M., Sellis, T., and Hanson, E. (1987). An Analysis of Rule Indexing Implementations in Data Base Systems. In L. Kerschberg (Ed.), *Expert Database Systems, Proc. from the First International Conference*, (pp. 465-476). Menlo Park, CA: Benjamin/Cummings.

Tarski, A (1955). A Lattice Theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics, 5*, 285–309.

VanEmden, M.H. and Kowalski, R.A. (1976). The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM 23*, 733–742.

Widom, J., Cochrane R.J., and Lindsay, B.G. (1991). Implementing set-oriented production rules as an extension to Starburst. *Proc. 17th VLDB Conf.*, (pp. 275–285). Barcelona, Spain.