Proceedings of the First International Conference on Information and Knowledge Management, Baltimore, MD, November 1992.

Approximate Knowledge-base/Database Consistency: An Active Database Approach[1]

P92-26

Leonard J. Seligman
The MITRE Corporation
McLean, Virginia

Larry Kerschberg
George Mason University
Fairfax, Virginia

Abstract

Many AI applications populate their knowledge-bases with information retrieved from large, shared databases. This paper describes a new approach to maintaining consistency between objects in a dynamic, shared database and copies of those objects which are cached in an application knowledge-base. The approach relies on an intelligent interface to active databases that we call a Mediator for Approximate Consistency (MAC). The MAC has several unique features: (1) it permits applications to specify their consistency requirements declaratively, using a simple extension of a frame-based representation language, (2) it automatically generates the interfaces and database objects necessary to enforce those consistency requirements, shielding the knowledge-base developer from the implementation details of consistency maintenance, and (3) it provides an explicit representation of consistency constraints in the database, which allows them to be queried and reasoned about. The paper describes the knowledge-base/database consistency problem and previous approaches to dealing with it. It then describes our architecture for maintaining approximate knowledge-base/database consistency, including techniques for specifying, representing, and enforcing consistency constraints.

Keywords: knowledge-base management, AI/DB integration, active databases, expert database systems, quasi-caching

1. Introduction

Recently, there have been many advances in the integration of artificial intelligence and database systems. For example, it is now common for commercial AI shells to provide tools to automatically retrieve schema information from a DBMS and help a knowledge engineer develop a mapping from that schema to objects in a knowledge-base (e.g., [Abarbanel86]). In addition, research prototypes such as BERMUDA [Ioannidis88] and IDI [McKay90] have demonstrated features which can enhance the performance of an AI/DB interface, including heuristic prefetching of database objects and preanalysis of rule preconditions to identify functionality (e.g., relational joins) which can be more efficiently performed within the DBMS.

Despite these advances, however, current approaches do not effectively support applications which must reason about the current state of dynamic, shared databases. An important limitation of these approaches is that they rely on caching data in an application's virtual memory but fail to address the issue of cache consistency adequately. To see why this can be a problem, consider a sample application, a knowledge-based Order of Battle Analyst, shown in Figure 1. The order of battle (OB) problem consists of analyzing a highly dynamic, multiple-source intelligence database and coming up with a current assessment of the enemy units on the battlefield, their strength, and the relationships among them. The timeline in Figure 2 demonstrates the problem. At t_3, the shared database is updated with information that is of critical importance to the OB application. Unfortunately, because there is no mechanism for ensuring cache consistency, the change is not propagated to the application. As a result, the conclusions made by the application at t_2, based on the outdated information, are now potentially flawed.

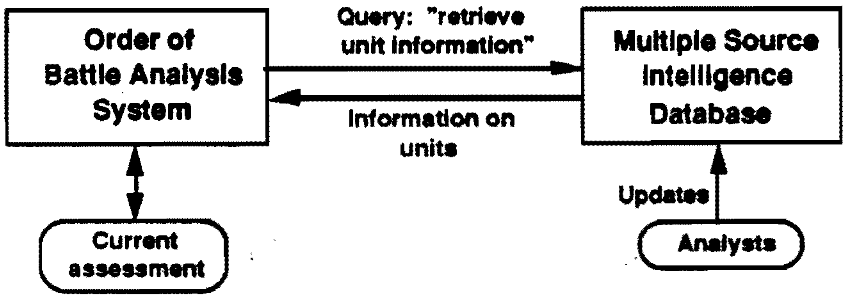

Order of Battle Analysis System
Query: "retrieve unit information"
Information on units
Multiple Source Intelligence Database
Current assessment
Updates
Analysts

Figure 1: Order of Battle example

Time	Event
t_0	Order of battle application queries the database to get information on enemy units
t_1	Application receives and caches information about UNIT-1, with type = "motorized rifle" and strength = 100
t_2	Application infers the presence of some higher level unit, its perceived mission, and its estimated overall strength, based on data cached at t_1
t_3	Database is updated with corrected or more current information about UNIT-1: type = "tank" and strength = 20. Some data cached in the application are now invalidated. In addition, conclusions inferred by the application at t_2 are potentially invalid.

Figure 2: Timeline for order of battle example

Active databases (e.g., [Stonebraker88, Sellis88, Delcambre88, Hanson89, McCarthy89, Widom90]), in which a forward-chaining production system is embedded in the DBMS, can address the cache consistency problem in two different ways.[2] First, for applications which can be implemented completely within the rule system of an active database, the problem does not exist. In an active database, the working memory of the production system is the dynamic shared database. As a result, there is no need for the rule-based application to create a separate copy of each data object about which it is reasoning. Unfortunately, because the inference engines of active databases are extremely primitive by AI system standards, this approach only solves the cache consistency problem for very simple rule-based applications [Stonebraker90]. Second, for knowledge-based systems that are too complex to be implemented entirely within a simple database inference engine, the rule processing capabilities of the active database can be leveraged for consistency maintenance. Coupling a knowledge-based system with an active database makes possible the specification of *alerter* rules in the database, which can be used to notify the knowledge-based system about changes to objects which have been cached in the application knowledge-base [Seligman90, Stonebraker90].

[2]For a good overview of active databases, see [Hanson92].

In the following section, we describe the limitations of these approaches and introduce our approach to knowledge-base/database consistency, including techniques for specifying consistency constraints. Section 3 describes our architecture. Section 4 describes our approach to representing and enforcing consistency constraints within the active database. Section 5 provides an illustrative example. Finally, we close with a discussion of the status of our work and likely future directions.

2. A New Approach to KB/DB Consistency

While the use of alerter rules in an active database does address the KB/DB consistency requirements of some applications, it fails to address the needs of many others. This is because many applications would be completely overburdened by receiving notifications about every update to any cached object. In addition, the message traffic generated by a large number of alerter rule firings could result in seriously degraded network performance. For example, a military deployment planning application might cache information about troops and equipment to be moved, the status of various ports and airports, and transportation resources such as airplanes, ships, and fuel. The development and refinement of a deployment plan is a time-consuming process which should not be interrupted continually by inconsequential database updates which might trigger unnecessary re-planning (e.g., that there has been a slight change in the amount of petroleum available at Airbase-27). On the other hand, certain updates might be significant enough to threaten the viability of a plan under development (e.g., that Airbase-27 has been attacked and will be unavailable for weeks). In such cases, particularly in crisis situations which require rapid responses to certain critical events, the application should be notified of the change in a timely manner.

Because of the need to maintain various kinds and degrees of consistency, which may be short of 100% consistency, a hardware model of cache consistency is not appropriate. What is required is a *quasi-cache*, as defined in [Alonso90]. Quasi-caches contain *quasi-copies*, which are cached copies whose values are allowed to deviate in controlled ways from the primary copies of those objects. Our work is the first we are aware of to consider the implementation of quasi-caches within the context of general purpose database management systems (using active database rules) and to apply quasi-caching to the problem of KB/DB consistency.

Quasi-caches and their consistency requirements are specified in our approach using a simple extension to a frame-based representation language. A declarative language is necessary for specifying and transparently enforcing consistency constraints so that the knowledge-base developer can be shielded from the low-level details of quasi-cache implementation within the active database.

To specify that an object should be cached, one creates a class which is a specialization of the class *DB-Class*. The definition for DB-Class is shown in Figure 3.

```
(Define-class   DB-Class
   (selection-conditions)    ; Defines the conditions under which a quasi-copy
                             ;   is added to the knowledge-base
   (selection-language)      ; Specifies the language to be used in specifying
                             ;   selection conditions
   (retraction-conditions)   ; Defines the conditions under which a quasi-copy
                             ;   is purged from the knowledge-base
   (consistency-condition)   ; Defines the conditions under which a knowledge-
                             ;   base quasi-copy should be refreshed
   (msg-priority))           ; Priority of knowledge-base update messages
                             ;   for this db-class
```

Figure 3: Definition of DB-Class

Five slots of DB-Class are used to define the consistency requirements of a quasi-cache: *selection-conditions, selection-language, retraction-conditions, consistency-conditions,* and *msg-priority*. We describe these in turn.

The *selection-conditions* slot contains a specification of one or more conditions which trigger the creation of a new instance of this particular DB-Class in the knowledge-base. Depending upon the value of the *selection-language* slot, this specification can either be in the Intelligent Database Interface Language (IDIL), a logic-based language used in [McKay90], or in a Lisp-like variation of SQL.

Retraction-conditions indicate when a quasi-copy should be purged from the quasi-cache. Valid values for retraction-conditions are the following:

- nil: This means never retract. This is the default, because it is the least expensive option. No retraction rules need be generated in the active database for this option.

- t: This means delete a quasi-copy whenever its selection-conditions become invalidated. Retraction rules must be generated in the active database to enforce this.
- <predicate>: This means that quasi-copies should only be purged from the quasi-cache when <predicate> becomes true. This option also requires the generation of retraction rules in the active database.

The *consistency-conditions* slot contains zero or more conditions which are used to specify when an update to the database should cause the corresponding quasi-copy to be refreshed. Valid condition specifications include the following:

- (version <n>): Refresh when the quasi-copy is more than n versions out of date
- (time <n>): Refresh when the quasi-copy is more than n minutes out of date
- (percent <attribute> <n>): Refresh when <attribute> varies by more than n percent from the cached copy
- (value <attribute> <n>): Refresh when <attribute> varies by more than n from the cached copy
- (member <attribute> <list>): Refresh when <attribute> is changed to a value which appears in <list>
- (<op> <attribute> <constant>): Refresh when the (prefix) expression evaluates to true. <op> can be one of '>', '≥', '<', '≤', '=', or '≠'.

The *msg-priority* slot is used to help manage the stream of update messages from the database to the knowledge-base. Proper use of this slot ensures that more critical knowledge-base updates will take place before less critical ones.

To define a new DB-Class, one uses the macro *define-DB-Class* , as shown in Figure 4. This figure shows the definition of a new DB-Class, TankUnit, which is also a specialization of Unit.[3] The IDIL expression in the selection-conditions slot indicates that instances of TankUnit are to be created in the application knowledge-base whenever there are instances of Unit and UnitAssets in the database such that Unit.type is "tank", UnitAssets.asset is "T-72", and Unit.name equals UnitAssets.uname.[4] The retraction-conditions slot indicates that instances are to be purged from the application knowledge-

[3]Our system uses multiple inheritance.

[4]"Ans" refers to the answer relation. Tuples that are returned into the answer relation are mapped into the specified slots (i.e., name, type, echelon, and strength). In IDIL, variables are preceded by an underscore character.

base only when Unit.type is changed to something other than "tank". There are two consistency-conditions shown in this example. First, a knowledge-base instance should be refreshed whenever it is more than three versions out of date. Second, it should be refreshed whenever the strength attribute changes by more than 30 percent from the currently cached value. Finally, the msg-priority slot indicates the priority of update messages from the active database to the application knowledge-base for instances of TankUnit.

```
(Define-DB-Class  TankUnit  (:superclasses  Unit)
   ;; Name, type, echelon, and strength are domain-specific slots
   (name)
   (type)
   (echelon)
   (strength)
   ;; The remaining slots are used to define the quasi-cache
   (selection-language  IDIL)
   (selection-conditions
      ;; Maps result of IDIL query into domain-specific slots
      ((ans  _name  _type  _echelon  _strength)
        <-
        (Unit  _name  _type  _echelon  _strength)
        (UnitAssets  _uname  _asset  _number)
        (=   _type  "tank")
        (=  _name  _uname)
        (=   _asset  "T-72")))
   (retraction-conditions  (≠  _type  "tank"))
   (consistency-conditions
      ((version  3)
        (percent  strength  30)))
   (msg-priority  5)))
```

Figure 4: An example DB-Class definition

3. An Architecture for Approximate Consistency Maintenance

Our approach for managing knowledge-base/database consistency relies on the use of an intelligent KB/DB interface that we call a Mediator for Approximate Consistency (MAC). The term "mediator" comes from [Wiederhold92] and refers to software that presents data at a higher level of abstraction for a higher layer of applications. The MAC abstracts away most changes to the underlying database and only reports those updates that the knowledge-base has defined as being significant.

Figure 5 illustrates the operation of the MAC. It is composed of two major submodules: the *translator,* which handles communication from the application to the active database, and the *mapper/message handler,* which handles communication from the active database to the application.

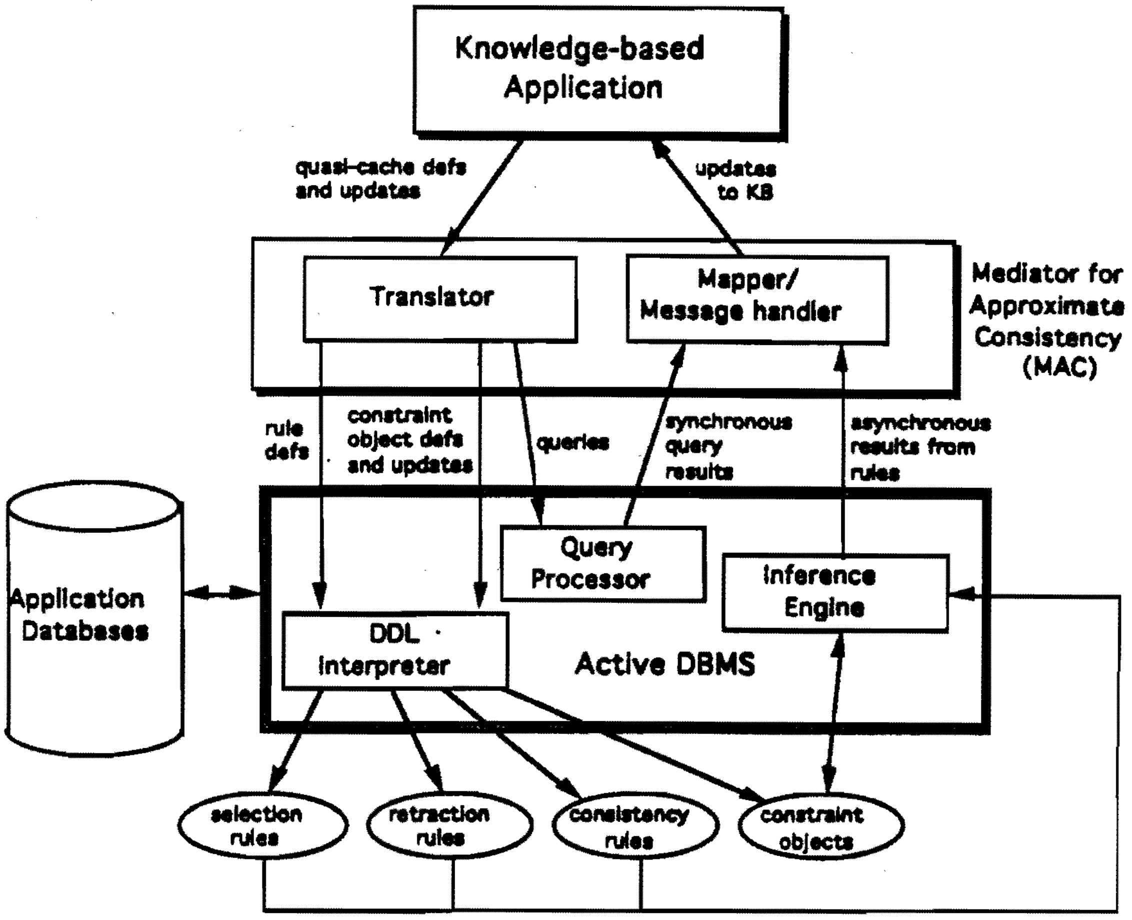


Figure 5: Architecture of the MAC

The translator accepts the declarative specification of a given class' consistency requirements as it appears in a DB-Class definition and translates it into the following: queries to be executed immediately, rules for monitoring the future state of the database, and data definition language commands which result in the creation of and updates to consistency constraint objects in the database. The queries which are to be executed immediately are used to populate the quasi-cache with those instances of the newly defined DB-Class for which the selection-conditions are satisfied at quasi-cache initialization time. The rules are of three types: *selection-rules,* which are used to monitor the database for future occurrences of the selection-conditions, *retraction-rules,* which are used to monitor the database for the retraction-conditions, and *consistency-rules*, which are used to monitor the database for conditions which require refreshing of quasi-copies in the knowledge-base. The purpose and representation of the constraint objects will be discussed in the following section.

The mapper/message handler receives notification of relevant database updates from the active database and maps them into the knowledge representation of the application. The mapper/message handler must accept two kinds of messages: synchronous query results, which are immediate responses to queries forwarded to the database by the translator, and asynchronous knowledge-base update messages, which result from the firing of selection, retraction, and consistency rules in the active database. Asynchronous messages are managed by a priority queue, to ensure that higher priority knowledge-base updates are processed before lower priority ones.

4. Constraint Representation

In our initial design, enforcement of consistency constraints was completely handled by rules in the active database. For each instance of a DB-Class in the knowledge-base which had a consistency-condition associated with it, there would need to be a separate alerter rule in the active database. For example, the DB-Class definition in Figure 4 has a consistency constraint that instances cached in the knowledge-base must be refreshed if their values for *strength* deviate by more than 30 percent from the current values in the database. Enforcement of this simple constraint would require one consistency rule per constrained instance. Assuming that <unit_1, tank, battalion, 50> and <unit_2, tank, battalion, 80> are two TankUnit instances cached in the knowledge-base, the rules shown in Figure 6 would enforce the constraint on strength for these instances.[5]

[5]All rule examples in this paper use a variation of POSTQUEL, the data definition and manipulation language of POSTGRES, a prototype active database system [Stonebraker88].

```
Define rule r_1
On replace to unit.strength
Where new.name = 'unit_1' and
   (new.strength < 35 or new.strength > 65)
Do
   Refresh the knowledge-base quasi-copy
   Delete and re-add this rule with the new appropriate boundary
     conditions

Define rule r_2
On replace to unit.strength
Where new.name = 'unit_2' and
   (new.strength < 56 or new.strength > 104)
Do
   Refresh the knowledge-base quasi-copy
   Delete and re-add this rule with the new appropriate boundary
     conditions
```

Figure 6: Enforcing consistency constraints using only rules

Our current design represents constraints explicitly as objects in the database, instead of burying the constraint representation in the *where* clause of consistency rules. This has two advantages over a strictly rule-based approach. First, it results in much less rule-base maintenance. Note that in Figure 6, every time a consistency constraint is violated, the rule must be deleted and readded with the new appropriate boundary conditions. This is much more computationally expensive than simply updating a couple of fields of a constraint instance object. Second, explicit representation of constraints is preferable to burying that knowledge in rules, because it permits the constraints themselves to be queried and reasoned about [Shepherd86].

Two classes of database objects are used to represent information on constraints. The first, a *constraint_def*, contains information on the definitions of constraints. A constraint_def is a six-tuple: $<c, \alpha, i, \rho, \tau, \varepsilon>$, where c is the database class which is constrained (e.g., unit), α is the constrained attribute (e.g., strength), i is an identifier for the application which is used for routing knowledge-base update messages to the application, ρ is the name of the rule which enforces this constraint, τ is the type of the constraint (e.g., version, time, percent, value), and ε is the permitted deviation beyond which knowledge-base quasi-copies must be refreshed. Instances of constraint_def are added when a new DB-Class is defined in the knowledge-base. The number of instances added by a given define-DB-class depends upon the number of consistency-conditions specified. The second class used to represent constraint information is the *constraint_instance*. Each constraint_instance contains information on the state of a particular quasi-copy and the specific consistency constraints on the corresponding object in the database. Figure 7 shows a partial schema for constraint_instance.

```
Constraint_Instance
   (constrained_obj,   ; OID of the constrained object
     attrib,           ; name of the constrained attribute
     constraint_def,   ; OID of the corresponding constraint_def
     KB_obj_name,      ; name of the knowledge-base object instance
     cached_val,       ; value for this attribute cached in the KB
     low_limit,        ; < this value means constraint is violated
     high_limit)       ; > this value means constraint is violated
```

Figure 7: Schema for the constraint_instance class

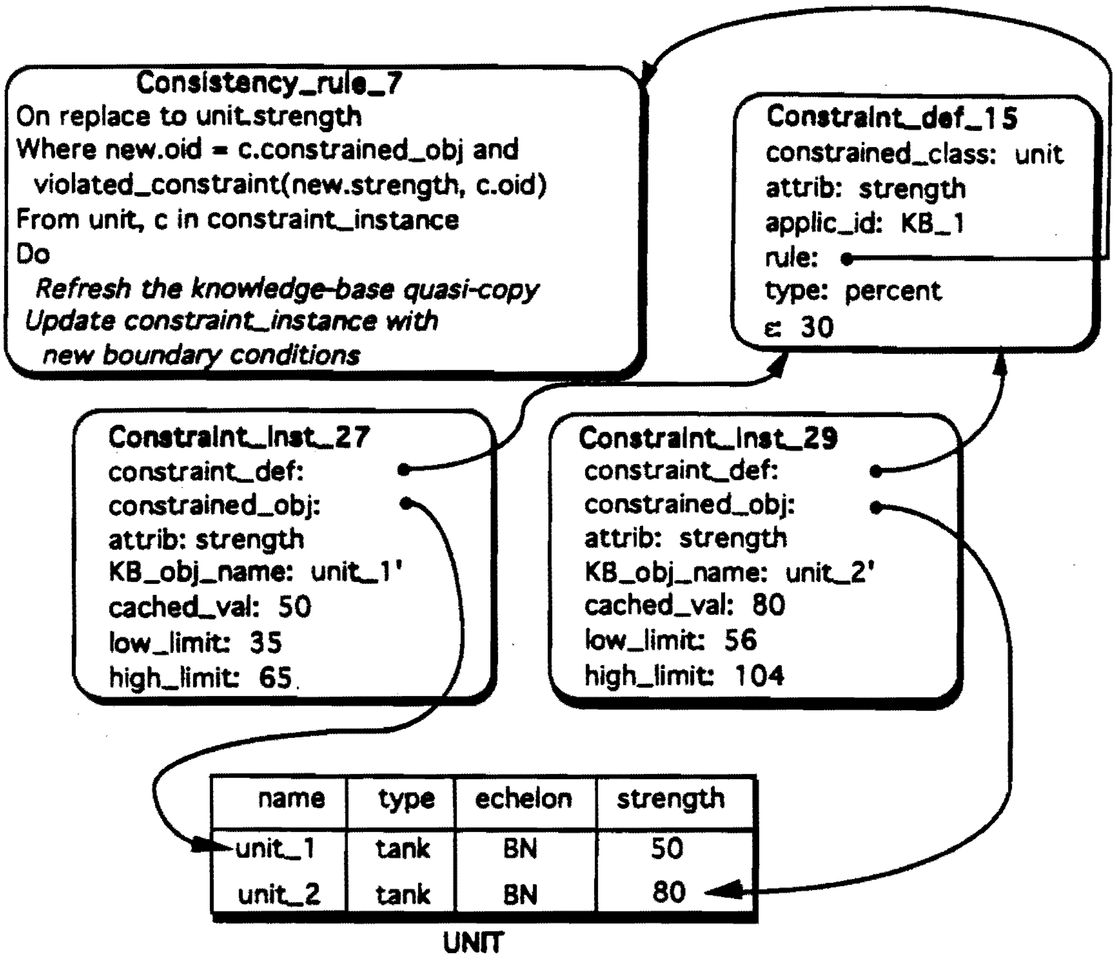
5. An Example

For this example, we will again use the DB-class definition for TankUnit in Figure 4. Evaluation of this definition would result in the generation of the following POSTQUEL query, which would be used to initialize the contents of the quasi-cache for TankUnit:

```
Retrieve (Unit.name, Unit.type, Unit.echelon, Unit.strength)
Where Unit.type = "tank" and
   Unit.name = UnitAssets.uname and
   UnitAssets.asset = "T-72"
```

In addition, selection and retraction rule definitions would be generated and sent to the database in order to monitor the database for future occurrences of the selection and retraction conditions.

Now suppose that as a result of the execution of the quasi-cache initialization query and the firing of selection rules, the knowledge-base now contains two instances of TankUnit, unit_1' and unit_2', with the following values: <unit_1, tank, battalion, 50> and <unit_2, tank, battalion, 80>. Figure 8 shows the state of the objects in the database which support maintenance of the TankUnit quasi-cache. There are two constraint instances, constraint_inst_27 and constraint_inst_29, each of which has a pointer to the definition of the constraint on the strength attribute, constraint_def_15. In addition, each constraint_instance has a pointer to the corresponding constrained object in the database, in this case instances of the class Unit. Constraint_def_15 contains a pointer to consistency_rule_7, which checks for violations of this constraint, and a reference to an identifier for the application (i.e., KB_1). This identifier is used to route update messages to the knowledge-base.



```
Consistency_rule_7
On replace to unit.strength
Where new.oid = c.constrained_obj and
  violated_constraint(new.strength, c.oid)
From unit, c in constraint_instance
Do
  Refresh the knowledge-base quasi-copy
 Update constraint_instance with
  new boundary conditions
```

```
Constraint_def_15
constrained_class: unit
attrib: strength
applic_id: KB_1
rule: •
type: percent
ε: 30
```

```
Constraint_inst_27
constraint_def: •
constrained_obj: •
attrib: strength
KB_obj_name: unit_1'
cached_val: 50
low_limit: 35
high_limit: 65
```

```
Constraint_inst_29
constraint_def: •
constrained_obj: •
attrib: strength
KB_obj_name: unit_2'
cached_val: 80
low_limit: 56
high_limit: 104
```

name	type	echelon	strength
unit_1	tank	BN	50
unit_2	tank	BN	80

UNIT

Figure 8: Database objects for enforcing consistency constraints

Consistency_rule_7 checks for constraint violations whenever the strength attribute of a constrained object is updated. Suppose for example that unit_1 has its strength updated to 30. Consistency_rule_7 would then check for constraint violations and would find that constraint_inst_27 had been violated. Since its preconditions would be satisfied, the rule would fire, resulting in an update notification to the knowledge-base for quasi-copy unit_1'. In addition, constraint_inst_27 would be updated to contain the new low and high limits of 21 and 39 respectively. No rule-base maintenance is required to process this update.

6. Conclusions

This paper has discussed a new approach to maintaining consistency between an application knowledge-base and related data in a dynamic, shared database. The techniques described here are applicable not only to knowledge-based applications, but to any application which could benefit from quasi-caching of shared data. To our knowledge, this work is the first to address quasi-caching within the context of general purpose database management systems.

We are currently developing a prototype implementation of our approach. We are implementing the MAC in the Common Lisp Object System (CLOS) and are providing an interface from the MAC to POSTGRES [Stonebraker88], a prototype extended relational database with rule processing capabilities. We do not anticipate that providing interfaces to other active databases would be a major difficulty, although we may have to change some of our underlying implementation assumptions (e.g., that every database instance has a unique object identifier). In addition, while we are using an extended relational database in our prototyping, there is nothing in our approach which would prevent us from using an object-oriented database, assuming it had adequate rule processing capabilities. That is why we have been careful to use the generic terms *class* and *instance* instead of their relational counterparts, *relation* and *tuple*.

Following the completion of our prototype implementation, we will use it to integrate an AI planning application with a POSTGRES database. During the course of developing the application, we expect to identify new requirements that will help us refine our design. Also planned is the construction of a simulation model that will allow us to assess when these techniques are more efficient than alternate ones, such as using alerter rules for all changes and periodic polling of the database to detect critical changes.

References

[Abarbanel86] R. Abarbanel and M. Williams, "A Relational Representation for Knowledge Bases", in [Kerschberg86].

[Alonso90] R. Alonso, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System", ACM Trans. on Database Systems, Vol. 15, No. 3, September, 1990.

[Delcambre88] L. Delcambre and J. Etheridge, "The Relational Production Language: A Production Language for Relational Databases", in [Kerschberg88].

[Hanson89] E. Hanson, "An Initial Report on the Design of Ariel", ACM SIGMOD Record, Vol. 18, No. 3, September, 1989.

[Hanson92] E. Hanson and J. Widom, "Rule Processing in Active Database Systems", in L. Delcambre and F. Petry, eds., The Emerging Landscape of Database and Information Systems, JAI Press, 1992 (to appear).

[Ionnidis88] Y. Ioannidis, J. Chen, M. Friedman, and M. Tsangaris, "BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine", in [Kerschberg88].

[Kerschberg86] L. Kerschberg, ed., Expert Database Systems: Proc. from the First International Workshop, Benjamin Cummings, Menlo Park, CA, 1986.

[Kerschberg88] L. Kerschberg, ed., Expert Database Systems: Proc. from the Second International Conference, Benjamin Cummings, Menlo Park, CA, 1988.

[McCarthy89] D. McCarthy and U. Dayal, "The Architecture of an Active Data Base Management System", Proc. of ACM-SIGMOD Int. Conf. on Management of Data, 1989.

[McKay90] D. McKay, T. Finin, and A. O'Hare, "The Intelligent Database Interface: Integrating AI and Database Systems", in Proc. AAAI-90, Boston, MA, July 1990.

[Seligman90] L. Seligman and L. Kerschberg, "On Active Databases: An Approach to Building Knowledge-base Management Systems", Proc. of Workshop on Knowledge-base Management Systems, AAAI-90, Boston, MA, July, 1990.

[Seligman91] L. Seligman and L. Kerschberg, "Active Federation: A New Architecture for Integrating AI and Database Systems," Proc. of Workshop on Integrating AI and Databases, IJCAI-91, Sydney, Australia, August, 1991. Also in L. Delcambre and F. Petry, eds., The Emerging Landscape of Database and Information Systems, JAI Press, 1992 (to appear).

[Sellis88] T. Sellis, C. Lin, and L. Raschid, "Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms", Proc. of ACM-SIGMOD Int. Conf. on Management of Data, 1988.

[Shepherd86] A. Shepherd and L. Kerschberg, "Constraint Management in Expert Database Systems", in [Kerschberg86].

[Stonebraker88] M. Stonebraker, E. Hanson, and S. Potamianos, "The POSTGRES Rule Manager", IEEE Trans. on Software Engineering, 14(7), July, 1988.

[Stonebraker90] M. Stonebraker, "Architectures for DBMS-Oriented Expert Systems", Workshop on Knowledge-base Management Systems, AAAI-90, Boston, MA, July, 1990.

[Widom90] J. Widom and S. Finkelstein, "Set-oriented Production Rules in a Relational Database Management System", Proc. of ACM-SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, May, 1990.

[Wiederhold92] G. Wiederhold, "The Roles of Artificial Intelligence in Information Systems", Journal of Intelligent Information Systems, Vol. 1, No. 1, eds. L. Kerschberg, Z. Ras, and M. Zemankova, Kluwer Academic Publishers, August 1992 (to appear).