TANGUY: Integrating Database, Rule-based and Object-Oriented Paradigms

B. Czejdo
Department of Mathematical Sciences
Loyola University, New Orleans
Ch. Eick, M. Taylor[1]
Department of Computer Science
University of Houston

Abstract

The database, rule-based and object-oriented paradigms are all relevant to many advanced applications, yet none of the three addresses all of the important issues. In this paper we present an extended data model called TANGUY, which integrates the three paradigms in order to capture the advantages of each. TANGUY provides automatic support for integrity constraints, exception handling and data-driven computations, relieving the application programmer of that burden. Our approach leads to much shorter and more reliable application programs, thereby greatly enhancing the productivity of programmers.

1. Introduction

The problems of integrating database, rule-based and object-oriented paradigms have gained popularity in current research. Most of these research efforts focus on the integration of object-oriented systems with databases (for survey see [20]). The major problem in merging object-oriented programming languages with database languages is to alleviate the impedance mismatch [20] arising when information must pass between two (syntactically and semantically) different languages [3, 8, 12, 13, 14, 18]. To tackle this problem, a variety of database programming languages have been proposed [2, 16, 17]. The problem of impedance mismatch, however, can be further alleviated by designing data models in such a way that they are directly compatible with the object-oriented languages.

Much less research has been performed that tries to integrate rule-based approaches into data- and knowledge bases. Intensive efforts have already been spent to support *active values*, sometimes also called *derived values*, in data- and knowledge bases. A derived value is an object whose value is computed by applying a function. Many advanced expert system shells such as KEE or ART already support active values [10, 11]. Furthermore, current database research even goes further by exploring the use of general declarative, deductive inference rules in addition to facts, and examines the problems of query processing and consistency enforcement in the context of these general deductive rules [1, 19].

The main direction of the work presented in this paper is to integrate the research efforts from the fields of databases, rule-based systems and object-oriented programming into a *knowledge base management system* (KBMS).

One of the main objectives we want to achieve by such an integration is to provide uniform interfaces to represent, access, and manipulate knowledge. For this purpose a data model has to be provided that is suitable to cope with objects that are data, operations/methods, production rules, and messages. This representation should allow separation of consistency enforcement and data-driven programming (e.g. triggers) from application programs — the KBMS takes care of many things the application programmer usually has to take care of. To enable the sharing and exchange of knowledge between application programs, we allow for permanent storage of knowledge that is of data, rules, operations and their associated methods that are retrievable using a uniform interface. Our model supports set-oriented interfaces, in addition to object-at-time interfaces, to facilitate the implementation of complex systems. It also supports the integration of data-driven, goal-oriented, and object-oriented programming, thereby facilitating the cooperation of software components relying on different programming paradigms within a single system.

More specifically, our approach extends the popular object-oriented programming language Smalltalk [9] to support production rules and the permanent storage of knowledge. In this paper, we will discuss how such an extension can be performed. Section 2 introduces the integrated data model TANGUY. Sections 3, 4 and 5 describe the model's support for queries, operations and rules, respectively. Section 6 concludes the paper.

2. The TANGUY data model

In order to integrate database systems, rule-based systems, and object-oriented systems, it is necessary to de-

[1] on leave from University of Keele, U.K.

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi
©World Scientific Publishing Co.

velop an extended data model that somewhat unifies the three approaches. A theoretical foundation of such a data model that includes inference rules has been given in [6]. In this section we shall present an extended data model called TANGUY, that supports object-oriented and rule-based programming, which is fully integrated with Smalltalk, and which provides a uniform interface to knowledge bases.

The concepts used in our data model are generally similar to those of other object-oriented database systems, but significantly different in the sense that some of the classes we support have instances that are sets of objects. The following concepts are essential for understanding the general principles of the TANGUY data model:

- *objects*, which are the most elementary components in the modeled universe of discourse.
- *object type*, which in the usual way allows objects to be classified.
- *object sets*, which are objects of the same type.
- *regular class*, whose instances are single objects.
- *set class*, whose instances are sets of objects.
- *class instance*, one of the members of the class.
- *maximal object set*, which consists of all instances of a (regular) class.
- *message*, which is sent to a class instance to execute a particular operation.
- *receiver*, for a given message, which is the class instance to which the message is sent.

In our approach, a knowledge base stores instances of classes with associated messages. For each object type, we have two classes: one is a regular class, whose instances are the individual objects of the type; the other is a set class, whose instances are sets of objects of the type. In all cases, messages are sent to class instances. This distinction allows us to send messages to sets of objects, which extends the capabilities of existing object-oriented languages, such as Smalltalk, in which messages have to be sent to a single object. Usually retrieval operations are most conveniently expressed in a set-oriented language. Set classes allow users to specify queries in terms of messages sent to sets of objects, thereby providing a powerful and high-level query language. On the other hand, certain operations (particularly updates) are often more properly associated with individual objects. Therefore, corresponding to each regular class, we have a set class whose instances are sets of objects. For each message associated with a regular class, we have a dual message associated with the set class.

As an example, let us consider the University knowledge base, whose schema is represented graphically in figure 1. It represents classes using rectangles, and messages using arrows from the receiver to the result class.

By convention, set classes are named by appending an 'S' to the end of the name of the corresponding regular class. In the schema, and throughout this paper, the class names are always in upper-case and message names are all in lower-case. Our graphical representation of the schema does not distinguish between regular and set classes, but represents both by one rectangle. The schema contains, among others, object classes COURSE, SECTION and GRADE.

The schema describes the classes for the benefit of users, listing the messages that can be sent to instances of each class. A knowledge base schema is also used by the system, to check the correctness of expressions and to correctly interpret application programs. By defining regular classes in the schema, we also implicitly define the corresponding set classes; e.g. the definition of COURSE implicitly defines the set class COURSES.

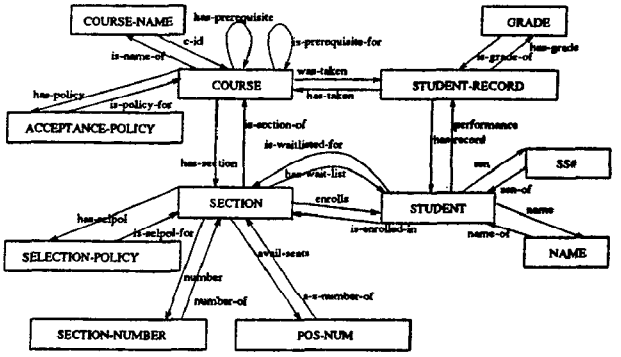

Fig. 1: Course Registration Schema

Our class hierarchy has a root class OBJECT, which has subclasses DATA-OBJECT, OPERATION and RULE. All other system- or user-defined classes have to be defined as subclasses of these classes. For example, SELECTION-POLICY is an operation class, and is therefore defined as a subclass of OPERATION. The instances of the class SELECTION-POLICY are operations, which, if applied to a set of students, select a single student ("the best") from this set. Different instances of the class SELECTION-POLICY may use different criteria to select a student, e.g. age, or grade point average.

Rule-classes are defined similarly as subsets of the class RULE. Examples of possible rule classes are 'the class of rules associated with data class SECTIONS', 'the class of rules associated with insert operations', 'the class of rules that need to be fired before performing some action', and 'the class of rules that need to be fired after performing some action'. A more detailed discussion on how to define and use rule classes is given in [7].

Our class hierarchy allows us to associate messages with data objects, with operation objects, or with rule objects, at essentially any level of granularity. This is important for providing accurate and efficient control of rules. Certain messages may even be associated with the class OBJECT, in which case they apply to all objects (data, rule or operation). Examples of such messages are 'insert' and 'delete' that create and delete instances of classes.

3. TANGUY Queries

Our user-interface accepts expressions in an object-oriented database language based on Smalltalk with special

extensions for rules, operations and set-oriented processing of knowledge. The basic constructs of the language are Smalltalk classes, instances of classes, and messages. The language requires several extensions to Smalltalk, however, in order to allow for knowledge base operations. The details of the query language are to be found in [5], but in this paper we shall summarize the most important features.

3.1 Database messages, selection and aggregation

The most elementary query would consist of a class instance name corresponding to the maximal object set for that class.

```
<simple-query> ::= <class-instance-name>
```

A typical simple query consists of a class instance name, followed by a sequence of messages. The following BNF statement extends the definition of simple query to allow messages:

```
<simple-query>::=<simple-query><db-message>
```

A database message is a message defined in the underlying database. For example, the following simple query finds the names of all students:

```
Student name
```

A simple query can also contain selection messages, which specify a subset of the current object set. Therefore the syntax of simple query can also be defined as:

```
<simple-query> ::= <simple-query>
<simple-selection-message>
```

A simple selection message is a keyword message (EQ, LT, GT, NE, etc.) that requires an argument that can be a constant or a simple query. GT and LT require one-element sets as the argument, and EQ and NE can accept any set as argument. An example of a valid simple query containing a simple selection message is is the following, which finds the number of taken seats on section number 100:

```
Section-number EQ:#(100) number-of taken-seats
```

The first message, EQ, is sent to the maximal object set for the class SECTION-NUMBER. This message is a keyword message, and therefore requires an argument which in this case is a one-element set containing '100'. The result of this operation is an instance of the class SECTION-NUMBER that is receiver of the next message 'number-of'. The last message in the above query is 'taken-seats', that is sent to the result of the message 'number-of'. The application of the message 'taken-seats' produces the requested number of seats taken.

Another example of a valid simple query is the following, which finds the number of seats taken on each of the sections with more than 50 students:

```
Section taken-seats GT:#(50)
```

Additionally, a simple query can contain an aggregate message, as defined below:

```
<simple-query> ::= <simple-query> <aggregate-
message>
```

Aggregate messages perform operations such as COUNT, SUM, AVERAGE, MIN, MAX, etc., on the current object set to yield a set containing a single (aggregate) numerical value. The following simple query finds the section number of the section which has the largest number of taken seats:

```
Section taken-seats max t-s-number-of number
```

The order of execution of the above sequence of messages is from left to right as discussed above.

3.2 Queries involving binary messages

We allow the usual set operations, i.e., UNION, INTERSECT and DIFFERENCE. The syntax of queries involving such operators is as follows:

```
<simple-query> ::= <simple-query>
<set-operator> <simple-query>
```

A query may consist of several simpler queries linked by set operators. The order of application of the set operators is made explicit by the use of parentheses. These set operations can be treated as binary messages.

For example, a valid query with a binary message is the following, which finds the names of those students who are both registered for section number 120 and wait-listed for section number 150:

```
((Section-number EQ:#(120) number-of enrolls)
INTERSECT (Section-number EQ:#(150) number-of
has-wait-list)) name
```

3.3 Defining new messages

The result of any simple query is a set of objects. A simple query can thus be viewed as a compound message to compute such objects. To facilitate the expression of subsequent queries, it is often convenient to create a new message (or operation) corresponding to the query. New instances of operation classes can be defined by using an operator *define-message*, which has to be sent to the operation class into which the new instance is inserted. This can be done by using the following syntax:

```
<message-definition> ::= <operation-class>
define-message:<name> receiver:  <class-name>
parameters:  <parameter-list>
result:  <class-name>
as:  [<sequence-of-messages>]
```

The above specifies that the defined message can be sent to any instance of the class specified after the keyword 'receiver', and the recipient instance should apply the messages specified after the keyword 'as'. The result of the message is an instance of the class specified after the keyword 'result'. A message definition may include a list of parameters, each of which must be named. The parameter list obeys the following syntax:

```
<parameter-list> ::= [<name>:  <class>
{, <name>:  <class>}*]
```

that is, there can be any number of named parameters, each specified as being an instance of a particular class. When the operator define-message is invoked, the new message is defined for each member of the receiver class, according to the expression given in the definition.

For example, we might define a new message to give the names of all students who are registered for a particular course. This message can be defined as follows:

```
OPERATION define-message:  student-names
receiver:  COURSE result:  NAME
as:  self has-section enrolls name
```

User-defined messages correspond to derived/active values in other approaches. If they are stored explicitly, they introduce redundancy into the knowledge base with consequent danger of inconsistencies occurring in the knowledge base. Alternatively, some systems choose to store only the definitions of derived attributes, but then they need to recompute the values each time a derived attribute is referred to. In TANGUY, the definitions of user-defined messages are stored in the operation base. It is also permissible to store the derived values explicitly in the knowledge base. In this approach, we maintain consistency by specifying rules which will automatically update the derived values in line with changes to related data, which will be discussed later.

3.4 Inheritance

A subclass automatically inherits the messages of its superclass, in the usual way for object-oriented systems. For instance, all classes inherit the messages associated with the class OBJECT. Further, all data classes inherit the messages associated with the class DATA-OBJECT. Similarly, operation classes and rule classes inherit the messages associated with the classes OPERATION and RULE, respectively. In addition, users can specify one class to be a subclass or superclass of another. For example, we might define a class GRADUATE-STUDENT to be a subclass of STUDENT. In that case, GRADUATE-STUDENT would automatically inherit all the messages of STUDENT, but might also have some messages of its own. We need a message linking the subclass to the superclass, but this message can be omitted from queries. For example, we can create a message 'is-a', with receiver GRADUATE-STUDENT and result class STUDENT. The inverse message 'graduate-spec' can be defined with receiver STUDENT and result class GRADUATE-STUDENT. To find the names of all graduate students, we could write

```
Graduate-student name
```

even though the message 'name' has receiver STUDENT rather than GRADUATE-STUDENT.

In case of multiple inheritance, it might happen that two superclasses have messages of the same name. We resolve this kind of conflict by explicitly specifying the superclass in the object-oriented query. In such a case, the previous query should be expressed as

```
Graduate-student is-a name
```

We also have an interesting case of automatic selection of subclasses, which is related to inheritance. This happens when we want to select a subclass from the current class, and apply a message to it. For example, suppose the message 'thesis-topic' is associated with the class GRADUATE-STUDENT. Now if we wish to know the thesis topic of all graduate students named 'Shah', we can express the query as follows:

```
Name EQ:#(Shah) name-of thesis-topic
```

The message 'graduate-spec' was unnecessary because there is no ambiguity in choosing the receiver for thesis-topic.

4. TANGUY Updates

Update operations have to be specified in terms of a small set of primitives. These primitives allow objects to be added to, or removed from, a class; and for addition, removal or modification of objects in a relationship with some other specified object. The primitives are defined on the class OBJECT, and all classes inherit them automatically. For example, to register a student for a section involves adding the student to the 'enrolls' relationship with the specified section. Similarly, to drop a student from a section involves removing the student from the 'enrolls' relationship with the section. These update operations can be specified as messages, whose receiver class is SECTION, and which take a student as parameter following a keyword. The only additional specifications required in the definition of an update are the checks which are specific to the update being defined. Before dropping a student from a section, we should check that the student has been enrolled for that section. Before registering a student, we should check that the student meets the requirements of the acceptance policy for the course. Therefore we define the messages as follows:

```
OPERATION define-message:  register
receiver:  SECTION
result:  ERROR-CODE parameters:  [s:  STUDENT]
as:
[(s  [self is-section-of has-policy])
   iffalse:  [ code for error-handling]
    iftrue:
        [(self is-full NOT)
         iftrue:[self insert-message:enrolls
         instance:s]
         iffalse:  [self insert-message:
         has-wait-list instance:s]]]

OPERATION define-message:drop
receiver:SECTION result:ERROR-CODE
parameters:  [s:  STUDENT]
as:
 [(s IN (self enrolls))
   if-true:[self delete-message:enrolls
   instance:  s]]
   [(self is-full not) and (self has-wait-list
   count > 0)
      if-true:  [self register [[self
      has-wait-list] [self has-selpol]]]]
    [(self is-too-small) and
    (University starts-soon)
      if-true:  (SECTION delete-instance
      instance:  self)] ]
```

In the above definition of 'register', we obtain the acceptance policy of the section's course, and apply this acceptance policy to the student given as parameter to the 'register' operation. The acceptance policy is an operation which returns a boolean result. The messages 'iffalse' and

'iftrue' also take parameters. These messages are sent to instances of the class BOOLEAN, and their parameters are actions which may or may not be carried out. The action given as parameter to 'iftrue' is carried out if the receiver of 'iftrue' is 'true'. Similarly, the action given as parameter to 'iffalse' is carried out if the receiver of 'iffalse' is 'false'. The operation returns a result which is an instance of the class ERROR-CODE.[2]. This result indicates whether the operation was performed successfully. Some other update operations, such as an operation which creates a new object, might return results of some other class (e.g., the object identity of the created object).

In the case of a drop-request, it is first checked if the student is enrolled in the particular section, if this is true the student is dropped from the section, possibly a student from the waiting-list is registered for the section, and possibly, if the semester starts soon and the section is almost empty, the section itself is canceled. The registration of wait-listed students is somewhat tricky: the section's selection policy is retrieved (using `self has-selpol`), and applied to the set of wait-listed students (which is the result of (`self has-wait-list`)) returning a single student, who will be enrolled in the section.

If the number of taken seats is stored explicitly in the database, the 'register' operation would need to be extended to increase the number of taken seats by 1 after a student has been successfully registered. We might also wish to specify exceptions, such as that the President's daughter is allowed to register for sections which are full, even if she does not satisfy the acceptance policy.

Thus it can be seen that the definition of an operation is liable to be quite long and complex, placing a heavy burden on application programmers. To overcome this problem, we represent both constraints and triggers as rules which are stored permanently and may be shared by many application programmers.

5. *TANGUY Production Rules*

The production rules, supported by the TANGUY data model, that differ significantly from traditional production rules, are called *activation pattern controlled rules*. They represent generalized forward chaining rules that have been tailored to be used in the context of object-oriented systems. Like any rule-based system, production rules consist of a left-hand side condition, which has to be a Smalltalk message that returns a boolean result, and a right-hand side action that consists of a sequence of arbitrary Smalltalk messages.

However, to tie the rule-based system better to the surrounding object oriented system, we assume that rules are only fired in particular *contexts*. Contexts are considered to be particular classes of messages; that is, requests to perform a particular operation with particular parameters. Every rule is assumed to have a *context-clause* that allows to restrict its applicability to particular messages; e.g. to messages sent to instances of a particular class, to messages calling particular operations, and/or messages that use a particular parameter. Context clauses are Smalltalk messages returning a boolean result connected by AND, OR, and NOT. In the case that the code of a context clause evaluates to true for the current message, the corresponding rule is taken into consideration to be fired; otherwise, it is excluded from the inference process.

Furthermore, activation pattern controlled rules are further subdivided into *before-* and *after-rules* depending on if they are fired before or after a particular message is executed. Additionally, two different *continuations* are provided for deciding what has to be done next after a rule has been fired. A continuation 'continue' continues the regular flow of control possibly firing other production rules, and a continuation 'return' that terminates the rule-based inference process, and transfers control back to the sender of the message with a particular result (frequently an error- or other message). *Priorities* can be assigned to rules, in order to cope with control problems: In the case that more than one rule is applicable, the rule with the highest priority is fired first.

We use the following syntax to define production rules in our system:

```
<rule-class> define-rule:  <rule-properties>
```

in which define-rule is a message whose receiver is a rule class. The effect is to create a new rule of this class. The 'define-rule' message also allows some parameters which are used to specify a name and other properties of the rule. These parameters obey the following syntax:

```
[rule-name:<rule-name>] context:(BEFORE // AFTER) <context-clause>
 condition:  <condition>   action:  <action>
 continuation:(CONTINUE // RETURN [<result>])
[priority:<integer>]
```

where the parameters within square brackets can be omitted if desired and '//' denotes either-or; e.g. a//b//c represents $\{\{a\},\{b\},\{c\}\}$.

The presence of production rules affects the behavior of the Smalltalk system as follows: If a particular message is called, first, the set of before-rules is computed that matches this message, and a forward chaining inference process is started for these rules; second, if this inference process terminates synchronously without firing any rule with continuation 'return' the message is executed; third, the set of after-rules that match the current message is computed and an inference process is started for these rules.

Like any other objects in the knowledge base, instances of rule-classes can be connected or deleted, and messages can be associated with rule instances. Furthermore, rules can be activated and passivated dynamically. Only active rules are fired by TANGUY's inference engine. The capability of passivating rules facilitates to support different versions of the same rule-based system — only the rules of the version that is currently run are active while rules belonging to alternate versions are still in the knowledge base in passive state. Furthermore, it is useful for coping

[2] We omitted the definition of the class ERROR-CODE

with exceptions, which will be discussed later.

As an example of a rule, we can specify that, if a section is not full and it has a non-empty waiting-list, a student should be selected from the waiting list and registered for the section. With each section is associated a selection policy, which determines how a student should be chosen from the waiting list. A selection policy is a procedure which selects one student from the waiting list. Different sections may have different selection policies. We discuss two different implementations of the above rules that illustrate how rules are tied to Smalltalk messages in TANGUY.

```
RULE define-rule:
 (rule-name:  register-waitlisted-students-v1
 context:  AFTER
 condition:  (mes.receiver belongs-to SECTION)
             AND (mes.receiver is-full not) AND
             (mes.receiver has-wait-list count>0)
 action:  [mes.receiver register [[mes.receiver
  has-wait-list][mes.receiver has-selpol]]]
 continuation:  CONTINUE)
```

The above command creates an instance of the class RULE named register-waitlisted-students-v1, which is an after rule. No context has been specified, which implies that the rule is considered to be fired after any message is executed. Let us assume that the message

```
se remove-message message:  enrolls
instance:  stu
```

is executed successfully. In the above se denotes a section, and stu denotes a student. The above rule is considered for execution, with mes.receiver bound to se. The rule's left-hand side checks if the object se is a section that is not full with an non-empty wait-list. If this is the case, the rule is fired registering up to three students from the waiting-list of the section, who are selected by the section's selection policy. On the other hand, the rule is not fired for messages sent to instances of other classes, because (mes.receiver belongs-to SECTION) evaluates to false in the latter case.

Although the above rule works correctly, it is inefficient in the sense that it is considered for execution after any message is executed — its context is empty, which specifies that it should be applied in any context. A more careful analysis reveals that wait-listed-students are only registered for a section, if a student is dropped from the section, or if the number of available seats is increased for this section. The rule register-waitlisted-students-v2 gives a more restrictive definition of the same rule.

```
(rule-name:  register-waitlisted-students-v2
context:  AFTER (mes.receiver belongs-to SECTION)
AND ((mes.operation = remove-message) AND
(mes.para.message = enrolls)) OR
(mes.operation = modify-message) AND
(mes.para.message = avail-seats) AND
(mes.para.new-instance > mes.para.old-instance))
condition:  (mes.receiver is-full not) AND
            (mes.receiver has-wait-list count>0)
action:  [mes.receiver register [[mes.receiver
has-wait-list [mes.receiver has-selpol]]]
continuation:  CONTINUE)
```

Let us illustrate, what happens after executing a message

```
se modify-message message:avail-seats
old-instance:44 new-instance:47
```

that increases the number of available seats by 3. In this case, mes.receiver is bound to 'se', mes.para.message is bound to 'avail-seats', mes.para.old-instance is bound to '44', mes.operation is bound to 'modify-message', and mes.para.new-instance is bound to '47'. For these bindings, the second part of the or-condition of the rule's context clause evaluates to true; therefore, the rule is considered for execution, registering wait-listed students for the section se, if possible. Note that in contrast to the first version the rule's left-hand side is only executed for a small subset of the possible messages.

Our next example rule specifies that, if a section is too small and the semester starts soon, the section is to be cancelled. We assume that a message 'is-too-small' is defined with receiver SECTION, and a message 'starts-soon' is defined with receiver the special class UNIVERSITY whose maximal object set contains only one object. The rule can be specified as follows:

```
(rule-name:  cancel-empty-sections
context:AFTER (mes.receiver belongs-to SECTION)
AND ((mes.operation = remove-message) AND
(mes.para.message = enrolls)) OR
(mes.operation = modify-message) AND
(mes.para.message = avail-seats) AND
(mes.para.new-instance > mes.para.old-instance))
condition:  (mes.receiver is-too-small) AND
(University starts-soon)
continuation:  CONTINUE))
```

As mentioned in section 3, instead of defining a message that computes the number of taken seats of a course from the enrolls message, we could also store this number explicitly in the knowledge base, maintaining this derived message every time a student adds for or drops from a section. The two rules given below, perform what we want:

```
(rule-name:  increase-#-enrolled-students
context:  AFTER (mes.receiver belongs-to SECTION)
AND (mes.operation = insert-message) AND
(mes.para.message = enrolls)
condition:  true
new-instance:  (mes.receiver taken-seats +1)
continuation:  CONTINUE)

(rule-name:  decrease-#-enrolled-students
context:  AFTER (mes.receiver belongs-to SECTION)
AND (mes.operation = delete-message) AND
AND (mes.para.message = enrolls)
condition:  true
new-instance:  (mes.receiver taken-seats -1)
continuation:  CONTINUE)
```

In the above rules +1 and -1 denote operations that add and subtract 1, respectively.

The rules introduced above are after-rules. However, sometimes it is desirable to perform rule-based computations prior to executing a particular message; for example, the enforcement of consistency constraints requires to check conditions prior to the execution of a particular message. The rule given below, enforces that only students that satisfy the acceptance-policy of a particular course (e.g. prerequisites) are allowed to register for a section offering this course.

```
(rule-name: enforce-course-acceptance-policy
context: BEFORE (mes.receiver belongs-to SECTION)
AND (mes.operation = insert-message) AND
((mes.para.message = enrolls) OR
(mes.para.message = has-wait-list))
continuation: RETURN)
```

It is important that the above rule carries the continution 'RETURN', returning control to the caller of the enrollment-request without executing the enrollment request, preventing the violation of the constraint. The next rule prevents that students are enrolled into sections that are full.

```
(rule-name: prevent-overbooking
context: BEFORE (mes.receiver belongs-to SECTION)
AND (mes.operation = insert-message) AND
(mes.para.message = enrolls)
condition: (mes.receiver is-full)
                instance: mes.para.instance)
continuation: RETURN)
```

Note that the last two rules enforce the constraints automatically. Application programmers do not need to write extra code for consistency enforcement, which results in significantly shorter application programs. Furthermore, the code for enforcing the constraints is only stored once in the knowledge base contrasting approaches [4, 15] which augment application programs by code to enforce consistency constraints. In the latter case, the code for enforcing a particular constraint is stored redundantly for every update operation that potentially violates the constraint.

Furthermore, different things might happen depending on the order in which the two above rules are fired. Consider for example, that a student wants to register for a full section of a course , for which (s)he does not have the necessary prerequisites; that is, the student's request violates the acceptance policy of the course. If the rule enforce-course-acceptance-policy is fired first, the enrollment request is immediately rejected. On the other hand, if the rule prevent-overbooking is executed first, the rule's left-hand side requests to wait-list the student for the section. This request is rejected by the rule enforce-course-acceptance-policy transferring control back to the right-hand side of prevent-overbooking (the caller of the wait-list request), which itself terminates with continuation return, rejecting the original enrollment request. That is, although different computations are performed for the two cases the result is the same, following the spirit of implementing rule-based systems in a parallel fashion so that the order in which rules are executed — if possible — does not affect the intended outcome.

However, in some situations it is necessary to sequentialize the execution of rules to meet particular application needs. Suppose, for example, that the president's daughter is allowed to register for section that are full, even if she does not satisfy the course's acceptance policy. The two rules, given below, take care of this situation:

```
(rule-name: b-register-president's-daughter
context: BEFORE (mes.receiver belongs-to SECTION)
AND (mes.operation = insert-message) AND
(mes:para.message = enrolls)
condition: (mes.para.instance
is-president's-daughter)
action: (RULES passivate rule-inst:
is-name-of)
priority: 1000
continuation: CONTINUE)

(rule-name: a-register-president's-daughter
context: AFTER (mes.receiver belongs-to SECTION)
AND (mes.operation = insert-message) AND
(mes.para.message = enrolls)
action: (RULES activate rule-inst:
is-name-of)
priority: 1000
continuation: CONTINUE)
```

In the above 'is-president's-daughter' is a boolean operation that returns true, if a particular object is a president's daughter. The above rule handles the exceptional treatment of daughters of presidents by associating a before-rule with enrollment messages that register a president's daughter that passivates the two consistency enforcement rules, and an after-rule that reactivates the two consistency enforcement rules after the president's daughter has been enrolled into a section. However, this implementation will only work correctly if the rule b-register-president's-daughter is executed prior to the two consistency enforcement rules, introduced earlier. This is achieved by assigning a higher priority to this rule (our system uses a default-priority of 0 for rules, if no priority has been specified). This example illustrates that the capability of assigning particular before- and after-computations to particular requests, facilitates to cope with exceptional cases in data- and knowledge bases.

In the presence of the rules introduced in this section that take care of exception-handling, consistency enforcement and data-driven computations, the code of the messages register and drop, introduced in section 4, degenerates to:

```
OPERATION define-message:register
receiver:SECTION result:ERROR-CODE
 parameters:[s:student] as:
[self insert-message:enrolls instance:s]

OPERATION define-message:drop
receiver:SECTION result:ERROR-CODE
 parameters:[s:student] as:
[self delete-message:enrolls instance:s]
```

which demonstrates the main advantage of our approach: due to the availability of general knowledge in the form of rules and of generic messages in the knowledge base that take care of general things with respect of a particular ap-

plication, application programs become extremely short, reliable, and easy to write, increasing a programmer's productivity significantly.

6. Conclusion

The paper presented an extended data model called TANGUY, whose objective is to integrate object-oriented programming with rule-based programming and permanent storage of knowledge in a Smalltalk environment. Our approach allows to separate knowledge that is general with respect to a particular application area from knowledge that is specific for performing a particular task. While special knowledge is still kept in application programs, the more general knowledge is stored permanently in a knowledge base so that it can be shared with other programs. Application programs are executed in the context of this general knowledge, which results in significantly shorter application programs because recoding of general knowledge is unnecessary in our system architecture. Furthermore, application programs tend to be more reliable because important tasks such as consistency enforcement or maintenance of derived values are performed by the knowledge base management system.

Finally, our system provides a set-oriented interface, which significantly extends the one-object-at-a-time capabilities of Smalltalk. Set-oriented interfaces also result in significantly shorter application programs because the programmer is relieved from sequentializing sets of operations into do-loops. Furthermore, the problem of finding an optimal order for executing a set of operations is no longer the programmer's duty but has become the responsibility of the knowledge base management system.

In summary we showed how the incorporation of rule-based and object-oriented concepts can greatly enhance database management systems, and thus better equip them to meet the requirements of advanced applications.

References

[1] A. Alashur, S.Su, and H.Lam, "A Rule-based Language for Deductive Object-Oriented Databases," in *Proc. Sixth Int. Conference on Data Engineering*, Los Angeles, 1990, pp. 58-67.

[2] A. Albano et al, "Galileo: a strongly-typed, interactive conceptual language", *ACM TODS* 10:2, 1985.

[3] T. Andrews and C. Harris, "Combining language and data base advances in an object-oriented development environment", *ACM conference on OOPSLA*, 1987.

[4] D. Cohen, "Compiling Complex Database Transition Triggers," in *Proc. ACM-SIGMOD Conference on Management of Data*, Portland, 1989, pp. 365-374.

[5] B. Czejdo, and M. Taylor, "Integration of Object-Oriented Languages and Database Systems in KOPERNIK," submitted for publication, Technical Report, University of Houston, June 1990.

[6] C. Eick, and T. Raupp, "Towards a Formal Semantics and Inference Rules for Conceptual Data Models," submitted for publication to Data & Knowledge Engineering, Sept. 1989.

[7] C. Eick, J. Liu, and P. Werstein, "Integration of Rules into a Knowledge Base Management System," in *Proc. First Int. IEEE Conference on Systems Integration*, Morristown, April 1990.

[8] D. Fishman et al, "Iris: an object-oriented database management system", *ACM TOIS* 5:1, 1987.

[9] A. Goldberg and D. Robson, "Smalltalk-80; the language and its implementation", Addison-Wesley, 1983.

[10] Inference Corporation, *ART Reference Manual*, Inference Corporation: Los Angeles, 1986.

[11] IntelliCorp, KEE Primer KEE Version 3.1. IntelliCorp publication number K3.1-P-1, 1988.

[12] W. Kim et al, "Integrating an object-oriented programming system with a database system", *ACM conference on OOPSLA*, 1988.

[13] C. Lecluse et al, "O2, an object-oriented data model", *ACM SIGMOD conference*, 1988.

[14] D. Maier and J. Stein, "Development and implementation of an object-oriented DBMS", in *Research directions in object-oriented programming*, MIT Press, 1987.

[15] L.Raschid, "Management of Disk Resident Rules", Technical Report, University of Maryland, 1990.

[16] L. Rowe and K. Shoens, "Data abstraction, views and updates in RIGEL", *ACM SIGMOD conference*, 1979.

[17] J. Schmidt, "Some high-level language constructs for data of type relation", *ACM TODS* 2:3, 1977.

[18] G. Shaw and S. Zdonik, "A query algebra for object-oriented databases", *International conference on data engineering*, 1990.

[19] P. Sheu, and W. Sull, "Knowledge management in deductive object bases," *Data & Knowledge Engineering*, vol. 5, no.1, pp. 29-58, 1990.

[20] S. Zdonik and David Maier (eds.), *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers: San Manteo, 1990.