Deductive Integrity Maintenance in an Object-Oriented Setting

Manfred Jeusfeld
Universität Passau
8390 Passau
Germany

Eva Krüger
Hörreuth 4
8391 Thyrnau
Germany

Abstract

The extension of integrity checking methods proposed for deductive relational databases to the case of object-oriented deductive databases offers new opportunities for more efficient consistency control: a reduction of the search space by finer granularity of updates, and a reduction of runtime integrity checking by incremental maintenance of the executable code generated for evaluating simplified rules and constraints within the database. Such an extended integrity system has been implemented in the KBMS ConceptBase.

This work was supported in part by the Commission of the European Community under ESPRIT Basic Research Action 3012 (CompuLog). Thanks to Matthias Jarke, Hendrik Decker and Gerhard Steinke for help and discussions. A version of this paper also appeared as MIP-9013, Universiät Passau, Germany, 1990.

1. Introduction

Today's database community is still searching for the next generation of database systems [ABD*89,ADV90]. The relational model is known for both its solid theoretical foundation and its expressive limitations. The background of mathematical logic induced the development of declarative query languages and integrity constraints accompanied by well-founded evaluation techniques. Deductive rules were proposed in order to overcome the limitations of relational calculus. Results by [DEC86,LST86,BDM88,KS88,BMM90] and others show that even in the deductive case integrity constraints can be checked efficiently. Examples for implementations of deductive databases are DedGin* [LV89] and DECLARE [KG90]. Nevertheless, they still inherit major disadvantages from the relational model:

1) Poor data modeling capabilities due to the fixed degree of aggregation in flat tuples.
2) Little support for application procedure development and reusability.
3) Little support for schema evolution due to maintenance.

Object-oriented database systems [MAN89] address these problems by providing an extensible set of data types or classes with attached procedures. Systems like HiPAC [DBM88], POSTGRES [SJGP90] and DAMOKLES [RRR*88] allow the developer to link procedures to certain events by so-called triggers, e.g., for activating integrity checking procedures after update events. One problem of such systems is the correctness of the imperative procedures in the case of cascaded trigger activations; another problem is their inflexibility concerning usage in different situations (or the need for the user to write a lot of triggers for each individual case).

The goal of this paper is to shift the technology of declarative formulas for integrity checking from the deductive relational to an object-oriented setting. Our experiments have shown that this shift implies both new problems and new opportunities for optimization.

The new problems concern understanding of the role of the class concept in integrity control, the need for efficient reconfiguration of complex objects in case of component changes, and the control of method applications. This paper concentrates on the first problem; the second problem is addressed in [GOCE90] while we solve the third one by embedding methods in envelopes that let them look like deduction rules [JJR89].

The new opportunities stem from the same sources as the problems. Firstly, practically all integrity checking methods for relational databases treat modifications of tuples as deletions followed by insertions; this means that too many constraints are checked. The aggregation abstraction of object-oriented databases allows us to associate specialized triggers to objects as small as individual attributes, thus providing much more precision in integrity optimization. Secondly, the explicit integration of methods allows the representation of triggered procedures directly in the database; however, in contrast to the OODB systems mentioned above, we preserve the idea of deductive relational algorithms that these triggers should be created automatically from predicative specifications. Thirdly, the classification abstraction of some object-oriented databases includes a metaclassing mechanism which can represent incremental schema modifications, in our case including constraint modifications. This allows us to shift the compilation of constraints into triggered procedures from update specification time to schema evolution time, and to preserve explicitly the relationship between triggers and their underlying constraint specifications, thus facilitating incrementality in optimization at the metalevel.

The rest of the paper explores these opportunities in two steps: a reinterpretation and slight extension of a relational method (section 2), and the application of this extension to the object-oriented case (sections 3 and 4).

Section 2 extends the well-known integrity checking method for deductive relational databases introduced by [BDM88] where a compilation phase for the insertion/deletion of constraints and rules, and an evaluation phase executing the generated code upon update events are distinguished. The code is derived from simplified forms responsible for update events on one of the literals occuring in the constraint. Particular properties of our version are the generation of simplified forms not only for constraints but also for deductive rules, and the incremental maintenance of the simplified forms.

Section 3 shows how we exploit the above observations to adapt the integrity checking method to the object-oriented case, using the knowledge representation language Telos and its implementation in the knowledge base management system ConceptBase as an example. A predicative language is related to the class schema of the database by defining so-called concerned classes for the literals, thus realizing the desired finer granularity.

Section 4 describes constraint management at the metalevel: how to maintain the declarative formulas, their executable forms, and the triggers as

part of the database. Updates on these objects can utilize the existing transaction mechanism of the database system. Moreover, representation within the database makes them subject to consistency themselves. For example, formulas refering to non-existing classes can be rejected because they violate referential integrity.

Section 5 summarizes the contributions of this work and outlines open problems.

2. Integrity Maintenance

This chapter presents the principles of the integrity checker in a way that is independent from a given data model. Therefore, the exploitation of specific properties of data models to locate data of a known type is left to the subsequent section. The integrity checker extends the relational checker proposed by [BDM88] by decomposing it into a part to be conducted at schema evolution time, and a part to be conducted at update time; also, by explicit specialization not only of integrity constraints but also of deduction rules.

2.1. Overview of the Method

Like all recent work in integrity control, the method is based on the well-known principles of simplification, first explicated by [NIC79]. Assuming that a database satisfies the integrity constraints before a transaction, possible violations must be caused by this transaction; this restricts the set of concerned integrity constraints to be rechecked. Moreover, the specific objects inserted or deleted by a transaction allow a specialization of the concerned constraints to the type of operation as well as to the specific objects in many cases. A simple model of transactions as a set of individual updates is assumed; we do not consider control structures in database transaction procedures. Besides that, the integrity checker improves run-time efficiency by subdividing its work into two phases: a compilation phase for new integrity constraints (and rules) independently from the database state and an evaluation phase which verifies transactions using the compiled formulas.

The user formulates an integrity constraint in a declarative, first-order language. The integrity checker automatically executes the following steps, starting with the compilation phase.

It transforms the formula into an internal normal form and generates parameterized simplified integrity constraints, each either responsible for insertions or for deletions. Such a simplified formula has to be evaluated only when information of a certain kind is inserted or deleted. The integrity checker limits the data affected by an integrity constraint by relating the literals of the constraint to the database schema. The stricter the classification of the data, the more efficiently does the integrity checker work.

The integrity checker stores the simplified constraints in such a way that they are quickly accessible if there is an update on a member of the corresponding data set. The integrity checker also inspects deductive rules with regard to integrity constraints. As in [DEC86], implicit information is only generated if it is relevant to some integrity constraint. Like the simplified constraints the generated simplified rules are stored close to the corresponding data. The integrity checker distinguishes how information enters into a rule. So the insertion of an element may lead to the implicit deletion of another one which has to be checked by an simplified integrity constraint, possibly after application of further rules.

At the end of transactions the integrity checker automatically executes the evaluation phase: For each element that is to be updated the integrity checker looks at the corresponding data collection (e.g., relation) for relevant simplified constraints or rules. It evaluates them together with the element to verify the transaction. If no error appears the integrity of the whole database is ensured. The integrity checker considers the insertion and deletion of integrity constraints and rules as normal transactions which may happen at any time during the life of a database. In the single-version database model considered in this paper (but not in the historical database setting actually supported by ConceptBase), a new integrity constraint is accepted only if the database satifies it.

2.2. Compilation Phase

An integrity constraint is a first-order predicative formula that has to be a theorem of each database state during which the constraint is considered valid. In essence, it can be seen as a continually asked Boolean query.

Deductive rules are formally a subset of the integrity constraints but they are made true by deducing the necessary conclusion instances. In both cases, the formula refers to the database state by literals. This makes such formulas especially attractive for relational-style data models since base and deduced literals correspond directly to base and deduced relations. The better the data addressed by a literal can be classified, the more the integrity checker may reduce the expense for maintaining integrity. The semantic of an integrity constraint or a rule is given by the canonical interpretation: true is exactly the information explicitly contained in the database or derivable with rules. This requires rules to be restricted; here we use stratified rules ([NT89]).

Definition 2-1

Let Λ be the set of positive literals of the declarative language, $L_1, \ldots, L_m \in \Lambda$. Then an **integrity constraint** is a well-formed formula *wff* in miniscope and disjunctive normal form matching one of the following patterns:

$$\exists x_1, \ldots, x_n \ (L_1 \wedge \ldots \wedge L_m \wedge R)$$

or

$$\forall x_1, \ldots, x_n \ (\neg L_1 \vee \ldots \vee \neg L_m \vee R)$$

Every variable $x_1, \ldots, x_n$ has to occur in at least one of the literals $L_1, \ldots, L_m$. The subformulas of R are either quantifier-free *wff*s or again in one of the above formats. If one of the variables $x_1, \ldots, x_n$ occurs in R then it is free. Variables other than the quantified ones are not possible.

Miniscope form means that each quantifier has a minimal scope. Disjunctive normal form includes that negations are pushed down to the literals. We restrict updates to ground literals as common for databases. But the algorithms for handling integrity also function for partially instantiated literals (see [BDM88]).

Definition 2-2

A **transaction** is a set of **updates**. Each update is a totally instantiated positive or negative literal over Λ[1]. A positive literal

[1] In fact, formulas also may be updated, see section 2.4.

L stands for the insertion of the information L, a negative literal $\neg L$ for the deletion of it. An integrity constraint is **relevant** to an update iff the complement of the update is unifiable with a literal in the constraint.

This relationship describes which integrity constraints have to be checked with an update: if a literal $\neg L$ (L) appears in a constraint then it has to be checked if a literal L' unifiable with L is an insertion (deletion).

Algorithm 2-1 [BDM88] determines a **simplified instance of an integrity constraint** IC relevant to an update U. Let L be a literal in IC unifiable with the complement of U. Variables in L that are universally quantified in IC but not governed by existential quantifiers are called **replaceable variables** of L.

Algorithm 2-1

- Instantiate in the constraint the replaceable variables of L according to the update U.
- Drop the quantifiers of the replaceable variables.
- Replace the literal L by $false$ if all its variables were replaceable.
- Possibly make simplifications.

To check the integrity of an update it is sufficient to prove the simplified instances of integrity constraints relevant to it. To be prepared for the efficient investigation of updates the integrity checker compiles a new integrity constraint to parameterized simplified constraints.

Definition 2-3

Let IC be an integrity constraint, L a literal occurring in IC, and v the set of replaceable variables of L. Then the triplet

$$(SIC, K, v)$$

is called the **parameterized simplified constraint** where K is the complement of L. SIC is generated as described in algorithm 2-1 but instead of the instantiation of the replaceable variables v these variables become parameters. K is called the **instantiation literal** of SIC.

A positive (negative) instantiation literal stands for a simplified constraint responsible for insertions (deletions). Note that the parameterized simplified constraint is built by simplifying a constraint with the complement of one of its literal occurrences. Having an update U one only has to take the parameterized simplified constraints with instantiation literals unifiable with U, to replace the parameters with the values given by U, and to evaluate the gained simplified instances of constraints. The following algorithm summarizes the compilation of an integrity constraint.

Algorithm 2-2

For each literal L of an integrity constraint IC:

- Generate and store the parameterized simplified constraint for L.
- Compile rules that have a conclusion literal unifying with the signless literal L (see below).

Deductive rules define how certain data are produced from other data of the database. These may be explicitly stored or again defined by rules. A rule may be recursive with the restriction that the conclusion literal is not allowed to enter negated into the body of its own rule, directly or indirectly using other rules. Deductive rules may formally be seen as a subset of the integrity constraints but they are made true by deducing the necessary conclusion instances.

Definition 2-4

A **deductive rule** is an integrity constraint matching the format

$$\forall x_1, \ldots, x_n \ (\neg L_1 \vee \ldots \vee \neg L_m \vee (\neg R \vee L_{concl}))$$

From definition 2-1 it follows that $x_1, \ldots, x_n$ are exactly the variables occurring in the **conclusion literal** L_{concl} and in $L_1, \ldots, L_m$. We usually write deductive rules in the form

$$\forall x_1, \ldots, x_n \ ((L_1 \wedge \ldots \wedge L_m \wedge R) \rightarrow L_{concl})$$

Every instantiation of the variables $x_1, \ldots, x_n$ that satisfies the subformula $L_1 \wedge \ldots \wedge L_m \wedge R$ also instantiates L_{concl} and leads to an implicit fact of the database. The following definitions set up a dependency network between rules [BDM88]:

Definition 2-5

A literal L_1 ($\neg L_1$) **directly depends** on the literal L_2 ($L_1, L_2 \in \Lambda$) iff there exists a deductive rule with a literal in its body that is unifiable with L_2 ($\neg L_2$), and this unification makes the conclusion literal identical to L_1. The transitive closure defines the relation L_1 **depends** on L_2.

For the purpose of integrity checking it is not necessary to consider a rule that can never contribute to the evaluation of a constraint, i.e. there is no instantiation literal of any simplified constraint that matches the conclusion literal of the rule or depends on it. During compilation phase the integrity checker transforms a concerned rule once to **parameterized simplified rules**. Simplification of a rule is done on its **condition expression**

$$\forall x_1, \ldots, x_n \ (\neg L_1 \vee \ldots \vee \neg L_m \vee \neg R)$$

similar to constraint simplification. All other rules whose conclusion literals are unifiable with one of the literals $L_1, \ldots, L_m$ or with one of the literals occuring in R also have to be compiled.

2.3. Evaluation Phase

When a new integrity constraint or a rule is added to the system, the formula passes the compilation phase once. The evaluation phase profits from this expense because it quickly finds the simplified parameterized constraints and rules that have to be evaluated in order to ensure integrity. Checking a transaction is very time-critical because of the availability of the system.

Algorithm 2-3

Inspect each update U of the transaction:
- Take each parameterized simplified constraint (SIC, K, v) whose instantiation literal K is unifiable with the update U:
  - Instantiate the variables of SIC that occur in K and are in v with the actual values of the update U.
  - Verify the simplified instance of a constraint against the database.
- In analogy proceed with each matching parameterized simplified rule:

- Partially instantiate the parameterized simplified rule with the update.
- By forward evaluation of this rule deduce all data and continue with them as normal updates.

If a literal L is to be inserted (deleted) then the integrity checker takes each parameterized simplified constraint (SIC, K, v) with a positive (negative) instantiation literal K unifiable with L. This constraint SIC is to be unified according to v and to be verified. If there is a parameterized simplified rule with a positive (negative) instantiation literal K' unifiable with L then also this rule has to be instantiated by the new literal and evaluated. This can deduce other literals C which are implicit insertions or deletions depending on the sign K' entered with into the rule. Then the integrity checker finds again a parameterized simplified rule or constraint with an instantiation literal unifiable with C (otherwise the first simplified rule wouldn't have been generated!). But this is only necessary if any literals have been deduced.

All integrity constraints and rules are evaluated on the temporarily updated database. If an integrity violation is detected a possible reaction is to refuse the transaction and to roll back the database to its old state. Another reaction is the creation of a parallel database version which is partially inconsistent but from which forward recovery can start without redoing the transaction. We do not discuss reactions to integrity violations further in this paper but note that more flexible recovery is possible with our approach using techniques borrowed from view updating.

2.4. Evolution of Constraints and Rules

A new integrity constraint must be completely evaluated against the database state (unless it can be proven directly inconsistent w.r.t. the existing database schema [BDM88]). If this doesn't cause an integrity error the constraint is transformed and compiled to simplified constraints as described above. When deleting an integrity constraint, the compilation also deletes all compiled forms of that constraint and updates the internal stored dependencies of rules applicable for checking this constraint. The insertion or deletion of rules is handled analogously; as mentioned before, simplified forms of rules are only generated if the created implicit information might violate some integrity constraint.

Algorithm 2-5

- If an integrity constraint IC is going to be inserted into the system:
  - Transform IC into the internal normalform.
  - Verify the complete constraint.
  - If integrity is not violated then compile the constraint.
- If a rule R is going to be inserted into the system:
  - Transform R into the internal representation.
  - If there exists a parameterized simplified constraint or a parameterized simplified rule with an instantiation literal that is unifiable with the positive conclusion literal of the rule R:
    - Deduce all conclusions of the rule R and treat them as normal insert updates.
    - If no integrity error appeared or there was nothing to check and if there exists an instantiation literal that is unifiable with the positive or with the negated conclusion literal of R then (and only then!) compile R.
- If an integrity constraint IC is going to be deleted then take each of its parameterized simplified constraints (SIC, K, v):
  - Delete each parameterized simplified rule where the instantiation literal K depends on the conclusion literal of the rule or the unsigned instantiation literal is unifiable with it, and if this rule is not necessary for the evaluation of any other integrity constraint.
  - Delete (SIC, K, v).
- If a rule R is going to be deleted and this rule was compiled by the integrity checker:
  - If there exists a parameterized simplified constraint or a parameterized simplified rule with an instantiation literal that is unifiable with the negated conclusion literal of the rule R:
    - Produce all deduceable literals with this rule and proceed with them as normal delete updates.
  - If no integrity error appeared or the rule didn't have to be evaluated then take each of the parameterized simplified rules of R:
    - Actualize compiled rules deducing literals the conclusion literal of R depends on.
    - Delete the parameterized simplified rule of R.

The integrity checker generates a parameterized simplified constraint for each literal of a constraint. It determines whether such a simplified constraint has to be checked for either insert or delete. If yes, the simplified constraint is related to the database structure in such a way that updates directly trigger its evaluation. Rules are handled analogously.

3. Adaption to an Object-Oriented Setting

The granularity of information in relational-style databases is fixed to tuples within relations. Therefore, changes to single components of a tuple have to be realized by updates of the whole tuple. The algorithms found in the literature simulate updates by deletion followed by the insertion of the revised tuple. By this, unnecessary evaluation of rules and constraints can happen as shown in the following example:

```
Relations:
  Drug(name:STRING;price:REAL)
  Patient(name:STRING;address:STRING;sex:CHAR;age:INT;takesDrug:STRING)
  Suffers(patient:STRING;symptom:STRING)
  Heals(drug:STRING;symptom:STRING)

Constraints:
  forall p,d,a,s,x,o
    Patient(p,a,x,o,d) and Heals(d,s) ==> Suffers(p,s)
```

If the address of a patient is changed, the constraint is evaluated though its truth is not affected.

Object-oriented databases offer a more flexible aggregation abstraction than relational databases. Implementations of object-oriented databases have different granularities for the smallest possible objects. In some cases, the decomposition of an object goes down to the level of attributes. This is necessary for relating attribute updates precisely to the affected constraints. Another requirement is that classes are regarded as objects. Updates on them (e.g., the insertion of a new constraint for a class) are treated as normal

database operations. Moreover, we assume that it is possible to reference attributes of a class in order to link the literals of a predicative formula to the class schema of the database. Finally, it goes almost without saying that we need a sublanguage for declaratively specifying deduction rules and integrity constraints as a starting point for our method.

One example of a language that fulfills these requirements is the knowledge representation language Telos [MBJK90]. After a short definition of the Telos data model a predicative sublanguage for deductive rules and integrity constraints is presented. Its literals are defined on top of the basic Telos data structure. Two assumptions are made which allow to determine a unique class (possibly an attribute class) concerning the truth of a literal occurence.

3.1. The Telos Data Model

Telos maps all relationships included in a (complex) object definition, esp. all its attributes, to a single data structure of so-called propositions:

Definition 3-1

Let ID be a non-empty set of identifiers. Then a finite subset

$$KB \subseteq \{Proposition(oid, x, l, y) | oid, x, l, y \in ID\}$$

is called a **Telos knowledge base** iff the *oid* component is unique within KB. The elements are referred as **propositions** or **objects**.

For the rest of the paper we assume that ID contains all finite character strings. The advantage of Telos are the uniformity ("everything is an object") and its simplicity. It is usual to denote Telos knowledge bases graphically where propositions are represented as links. An example is given in fig. 1. The propositions corresponding to the 11 graphical items are:

```
Proposition(Person,Person,-,Person)
Proposition(Patient,Patient,-,Patient)
Proposition(#3591.18,Patient,*isa,Person)
Proposition(Drug,Drug,-,Drug)
Proposition(#3591.20,Patient,takes,Drug)
Proposition(Fenta_forte,Fenta_forte,-,Fenta_forte))
```

```
Proposition(#3591.101,Fenta_forte,*instanceof,Drug)
Proposition(Jack,Jack,-,Jack)
Proposition(#3591.104,Jack,*instanceof,Patient)
Proposition(#3591.107,Jack,drug1,Fenta_forte)
Proposition(#3591.108,#3591.107,*instanceof,#3591.20)
```

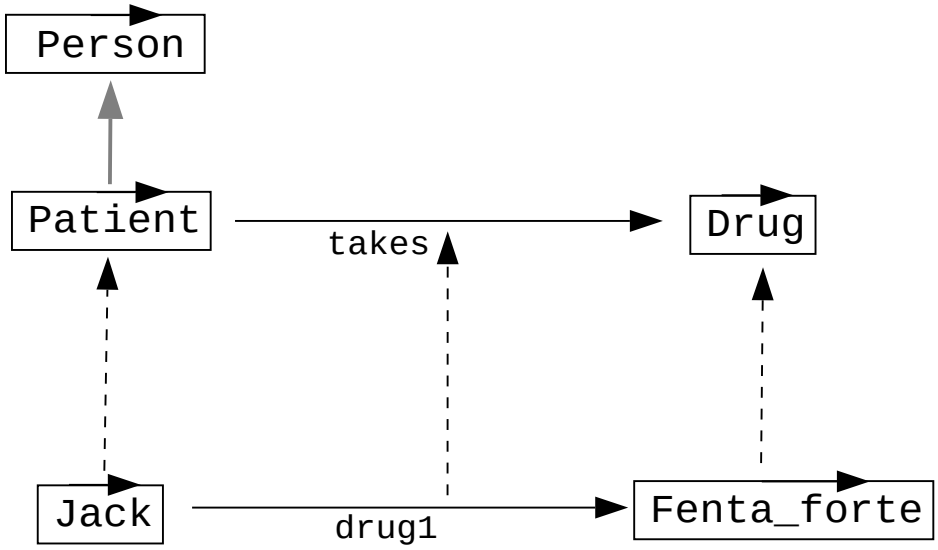

Fig. 1: Example of a Telos knowledge base

So-called **individual** propositions with identical first, second and fourth component are drawn as nodes. The other objects are called **attributes** with two special cases: attributes with third component (label) `*instanceof` are called **instantiation attributes** and attributes with label `*isa` are called **specialization attributes**. For propositions of the form

```
Proposition(oid,x,*instanceof,c),
Proposition(oid,c,*isa,d)
```

x is called **instance** of c (c is the **class** of x) and c is called the **subclass** of d (d is a **superclass** of c). In the following we assume that a Telos knowledge base KB contains the objects `Proposition`, `Individual`, `Attribute`, `InstanceOf` and `IsA` which have as instances all propositions of KB that have the form indicated by the class name. The structural properties of instantiation, aggregation and specialization are defined as built-in rules and constraints (see section 4).

3.2. Embedding a Predicative Sublanguage

The fixed knowledge base schema of 4-ary propositions[2] is not used directly for modeling purposes like e.g. in the relational case. Instead, classes serve in this role. Telos classes are not distinguished from other objects. Particularly, classes may themselves be instances of other (meta-) classes. This flexibility is useful for modeling but bears the danger of having no structure or "meaning" one can reason on. Therefore, Telos relies on a predicative language. It is made up from the following constructors:

```
forall x/C F        exists x/C F
F1 and F2           F1 or F2          F1 ==> F2   not F
AttrValue(v,l,w)    InstanceOf(v,w)   IsA(v,w)
TRUE                FALSE
```

Formulas refer to a Telos KB by using the three literals `AttrValue`, `InstanceOf`, `IsA` or by typed quantifications. The types of the quantified variables are interpreted as their classes, i.e. they are shorthand for the formulas:

```
forall x not InstanceOf(x,C) or F
exists x InstanceOf(x,C) and F
```

As a consequence all formulas can be transformed to equivalent range-restricted formulas as given by definition 2-1. Note that each Telos KB is finite so that infinite ranges are not possible. The interpretation of formulas follows classical logic as in [BDM88]. Deductive rules are mapped to function-free Datalog-neg programs for the purpose of query evaluation [STAU90].

3.3. Limiting the Search Space

A key issue for the success of the simplification method presented in section 2 is to limit as much as possible the search space of constraints and rules to be evaluated for a given update. In the relational case, one way to achieve this is to have as many relations as possible in the schema. Then, all

[2] Additional temporal components present in the full Telos language [MBJK90] are left out in this paper for a more concise presentation.

simplified parameterized formulas whose instantiation literals are different from those of the update are not in the search space just because the names of the literals do not match. In that sense, the worst case is a single universal relation where only the sign of a literal and its parameter instantiation in a constraint (or rule) can prevent the system from evaluating the simplified form.

Telos pays for its flexibility by falling into the worst case. We have a single "base literal" `Proposition` and a fixed set of three derived literals given by the following definitions:

```
forall v/Proposition forall w/Proposition forall p/InstanceOf
  Proposition(p,v,*instanceof,w) ==> InstanceOf(v,w)

forall v/Proposition forall w/Proposition forall p/IsA
  Proposition(p,v,*isa,w) ==> IsA(v,w)

forall v/Proposition forall w/Proposition forall l/Proposition
  (exists p/Attribute exists PP/Attribute exists l1/Proposition
   exists o/Proposition exists VV/Proposition exists WW/Proposition
     Proposition(p,v,l1,w) and
     Proposition(o,p,*instanceof,PP) and
     Proposition(PP,VV,l,WW))
  ==>
     AttrValue(v,l,w)
```

If the above definitions were treated as deductive rules each update of a `Proposition` literal would potentially trigger a simplified instance of each of the three rules. Only non-matching parameters could then prevent rule evaluation. This situation is unacceptable since all Telos transactions contain proposition updates. On the positive side, Telos as an object-oriented language offers the notion of classes to define structure within the KB. Two observations regarding classes make them a good candidate for limiting the search space of a literal:

Observation 1

Each Telos object has at least two classes: `Proposition` and one of `Individual`, `Attribute`, `IsA` and `InstanceOf`.

Observation 2

> Each variable in a predicative formula is bound to a class. Thus, instantiation to a variable corresponds directly to the notion of instances in object-oriented languages.

The problem now is to find a class for each literal occurrence of a formula that is as small as possible, i.e. a class that has a minimal number of instances which govern the solution of the literal.

Definition 3-2

> Let KB be a Telos knowledge base and L be a literal occurrence of a predicative formula. Then a **concerned class** c for L is a Telos class with the following property: inserting or deleting an instance of c can affect the truth of L in KB.

By observation 2 and the definition of the three Telos literals the set of concerned classes is never empty: the class `Proposition` always applies. In order to have a finer range we demand the following property for Telos knowledge bases:

Assumption 1

> Referential integrity is guaranteed for every Telos knowledge base, i.e. if `Proposition(`oid`,`x`,`l`,`y`)` is in the KB then there are propositions with first components x, l and y in the KB.

This assumption is not a real restriction since it is guaranteed by the structural axioms of Telos (see section 4). With assumption 1 and the definition of the literals it follows that one can take the class `InstanceOf` for the `InstanceOf`-literal: `p` may never be part of a Telos KB if `v` or `w` are missing. Thus, the literal `InstanceOf(v,w)` succeeds exactly for the instances of class `InstanceOf`. Analogously, class `IsA` can be taken as the concerned class of specialization literals. If the second argument in `InstanceOf(v,w)` is instantiated then one can take this object `w` as the concerned class.

The interesting case is the `AttrValue` literal due to the three condition literals and its numerous quantifications. If all variables are free in such a literal then nothing better than `Attribute` can be taken as a concerned class. Since attributes make up a major portion of a Telos knowledge base this is not satisfactory. A second restriction enables `AttrValue` literals to be connected to "user defined" attributes like the `takes` attribute in fig. 1.

Assumption 2

> If a literal `AttrValue(`x`,`l`,`y`)` occurs in a formula then l must be instantiated and the number of KB propositions unifying with `Proposition(`Oid`,`c`,`l`,`Y`)` is exactly one. Oid and Y are variables and c is equal to x if x is instantiated, otherwise, c is equal to the type of the variable x in the formula (observation 2).

This assumption enforces the existence of a unique attribute in the KB that corresponds to the literal. With a Telos KB fulfilling it one can take the unique solution for the object Oid in assumption 2 as a concerned class.

The concerned classes enable the simplified forms to be found quickly when an update takes place. The more subclasses are defined and used in predicative formulas, the smaller is the number of instances to be expected for the concerned classes. Implementations of Telos (and of many other object-oriented databases, e.g. Iris [LHW90]) promote the idea of extensibility through updates to the set of classes in the database. Thus, poorly balanced class hierachies can be corrected at run-time of the system improving performance of integrity checking.

3.4. Example Session of ConceptBase

The knowledge base management system ConceptBase [EJJ*89] has been developed for purposes of requirements modelling and support of design activities [JMSV90]. Its data model is the knowledge representation language Telos. ConceptBase is implemented in a client/server architecture using UNIX inter-process communication with user interfaces for SunVIEW and X11. The kernel server is mostly written in Prolog and the user interfaces mostly in C.

The screendump in fig. 2 shows a graph browser loaded with classes and instances of a KB on patients. Patients are persons who can suffer from certain symptoms and may take drugs. Drugs consist of agents that work on symptoms. A patient Jack takes the drug Fenta_forte which contains Fentanyl. This agent is used to reduce his fever symptom. The symptom pain has no relationship to the Fentanyl or Jack.

In the frame editor at the upper right corner of the screen a deductive rule defines that a drug heals a symptom if it contains an agent that has positive effects on the symptom. A constraint labelled `minimalAgents` demands that a patient may not take a drug that heals a symptom he doesn't suffer from. The last component `Always` of the literals is due to the temporal feature of Telos and not necessary for the purpose of this paper.

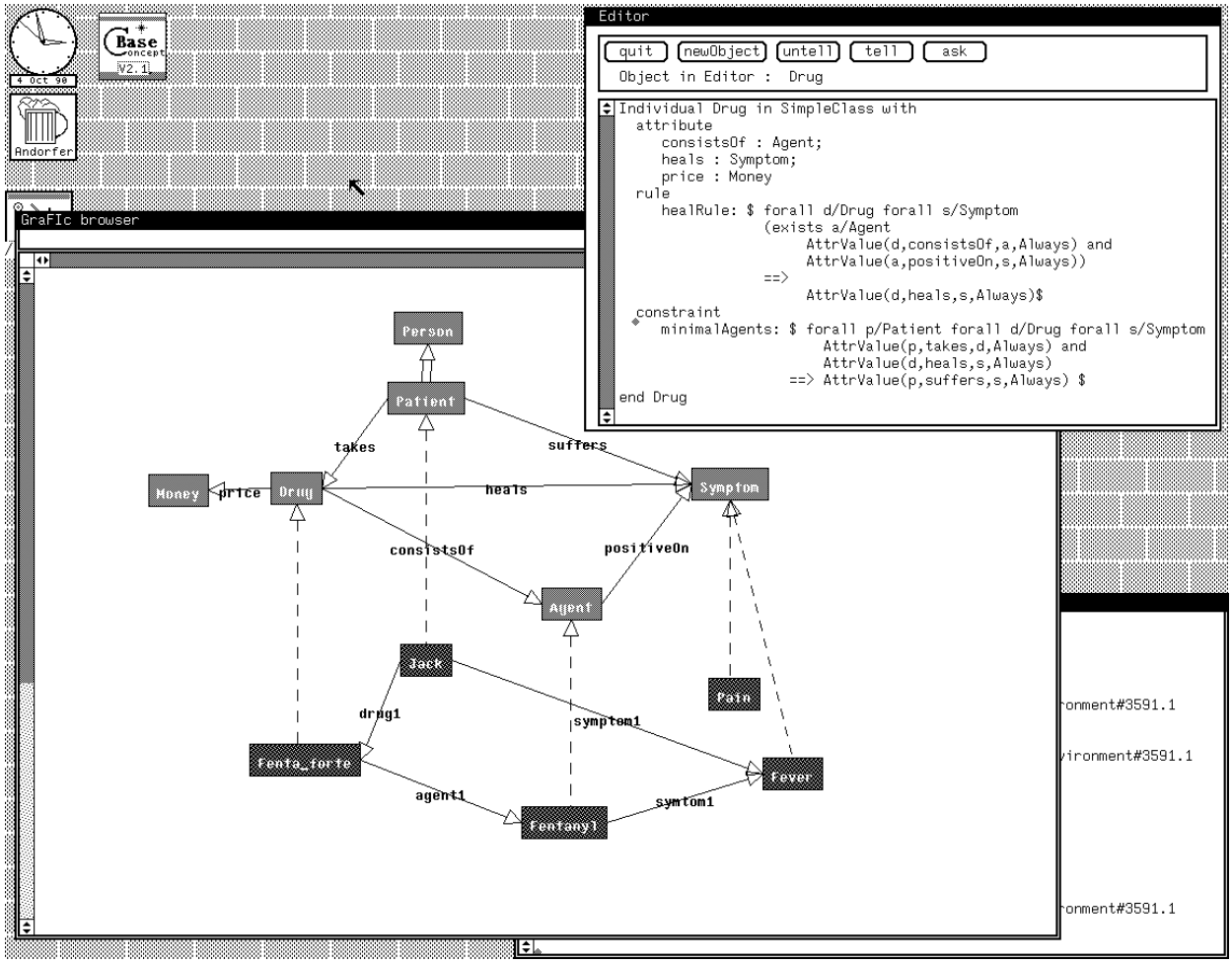

Fig. 2: ConceptBase session showing classes and instances around `Patient` and definition of two formulas on it

The screendump in fig. 3 shows the behavior of ConceptBase when inserting an object that violates an integrity constraint. In this case, a second instance of the `positiveOn` attribute for the agent Fentanyl is inserted. The literal corresponding to the new attribute is

```
AttrValue(Fentanyl,positiveOn,Pain)
```

which triggers a simplified form `#3591.77` of the `healRule`. A new solution

```
AttrValue(Fenta_forte,heals,Pain)
```

is derived and inserted into the corresponding simplified form `#3591.58` of the integrity constraint `minimalAgents`. Since the patient `Jack` takes `Fentanyl` and does not suffer from pain an error message is generated and the transaction is rejected. Note that an update of an attribute that is not

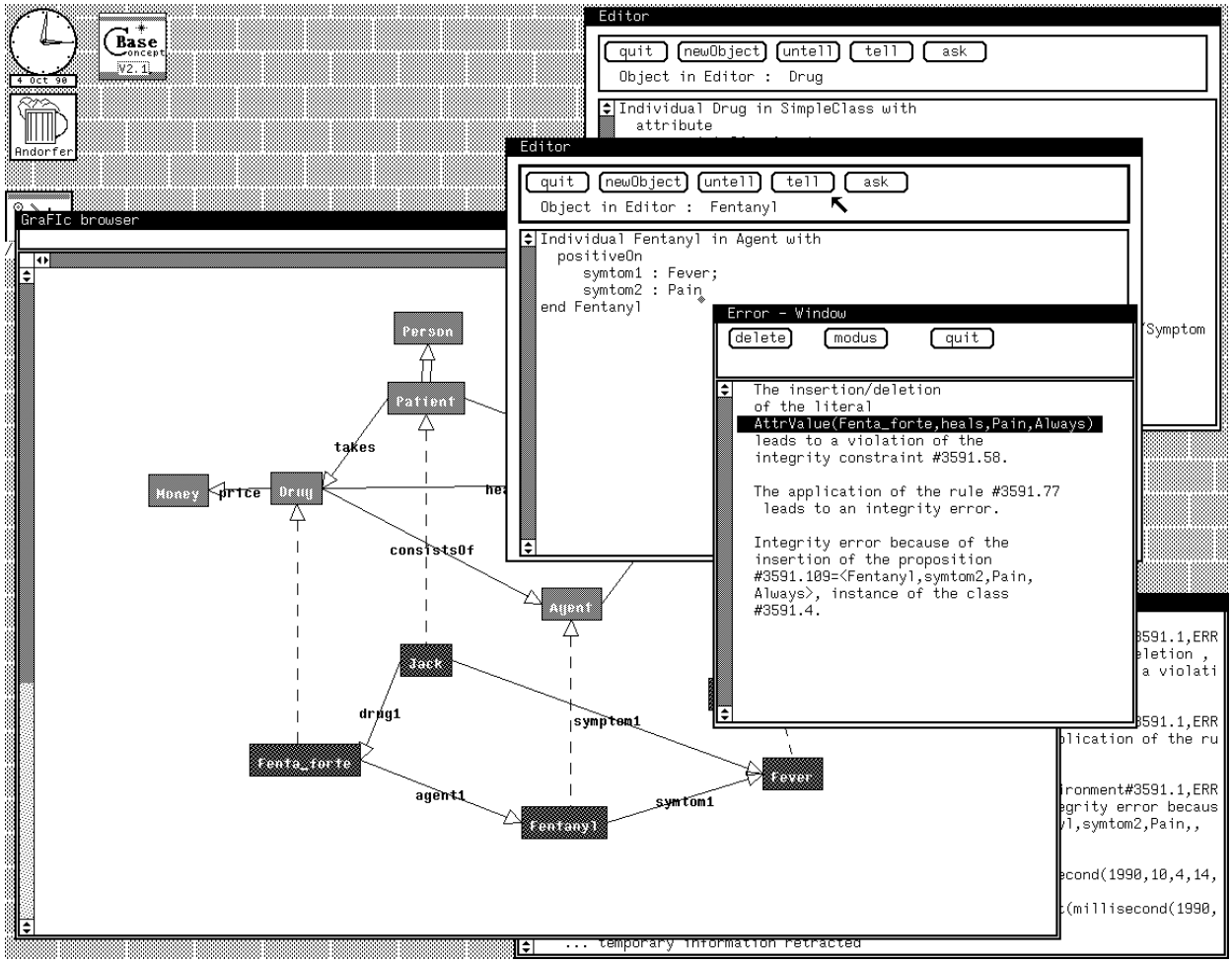

Fig. 3: ConceptBase after insertion of a new symptom for `Fentanyl`

concerned by literals occuring in rules or constraints (e.g. the price attribute for drugs) will not trigger integrity checking.

4. Towards a Consistent Information System

Building an information system that implements Telos requires a way to store a Telos knowledge base and a sub-system for handling the predicative language. Such formulas are statements about some world [BL86] like any proposition of a Telos KB. Thus, it is natural to represent them as part of the KB, i.e. as propositions. A set of six predefined formulas defining consistent aggregation, instantiation and specialization together with the objects

they refer to (`Proposition`, `InstanceOf`, `IsA`, `Individual` as well as others) are assumed to be included in each Telos knowledge base [KMSB89]. If a Telos KB satisfies the six language axioms it is called **structurally** consistent. For instance, referential integrity is ensured. In the current version of ConceptBase, structural consistency is checked by hand-optimized procedures located in the storage sub-system.

Section 2 provided algorithms for automatically generating and evaluating code for declarative rules and constraints. The information system has the task to react to insertions and deletions of such different objects as rules, constraints and literals by executing the right portion of the code, i.e. the code has a certain relationship to KB updates and should be stored within the KB. Fig. 4 presents a model that is able to capture these relationships uniformly and can be used to model and control the implementation of predicative formulas.

At the top layer the class `Proposition` stands as a synonym for Telos KBs since all propositions are instances of it. Operations apply to propositions, e.g. compiling a predicative formula. Execution is done by tools. Instantiating the top level objects can be interpreted as the **specification** of a certain system architecture, in this case the `ICcompiler` of ConceptBase. It offers an operation called `compile` which takes an `IntegrityConstraint` and produces as output the evaluable `SimplifiedForms`.

The bottom (instance) level is interpreted as **execution** of the system, e.g. in fig. 4 the compilation of a new constraint on patients and drugs. The dependency of the generated code to the original text of the constraint is recorded by an instance of the object `compile`.

The simplified constraints are evaluated by a meta-interpreter called `ICevaluator`. Inserting a new integrity constraint corresponds to adding the capability for checking it (and its simplified forms) to the evaluator tool. Fig. 5 shows how this is represented in the tool model. The `ICevaluator` now has as a new operation the ability to check the simplified constraint of fig. 4 for the insertion of `heals` literals. The simplified constraint is instantiated by applying the *mgu* of `AttrValue(_d,heals,_s)` and `AttrValue(Fentanyl,heals,Pain)` to it.

The tool model is accompanied by generic procedures that trigger messages of the form

```
Tool: by Operation from=Proposition
```

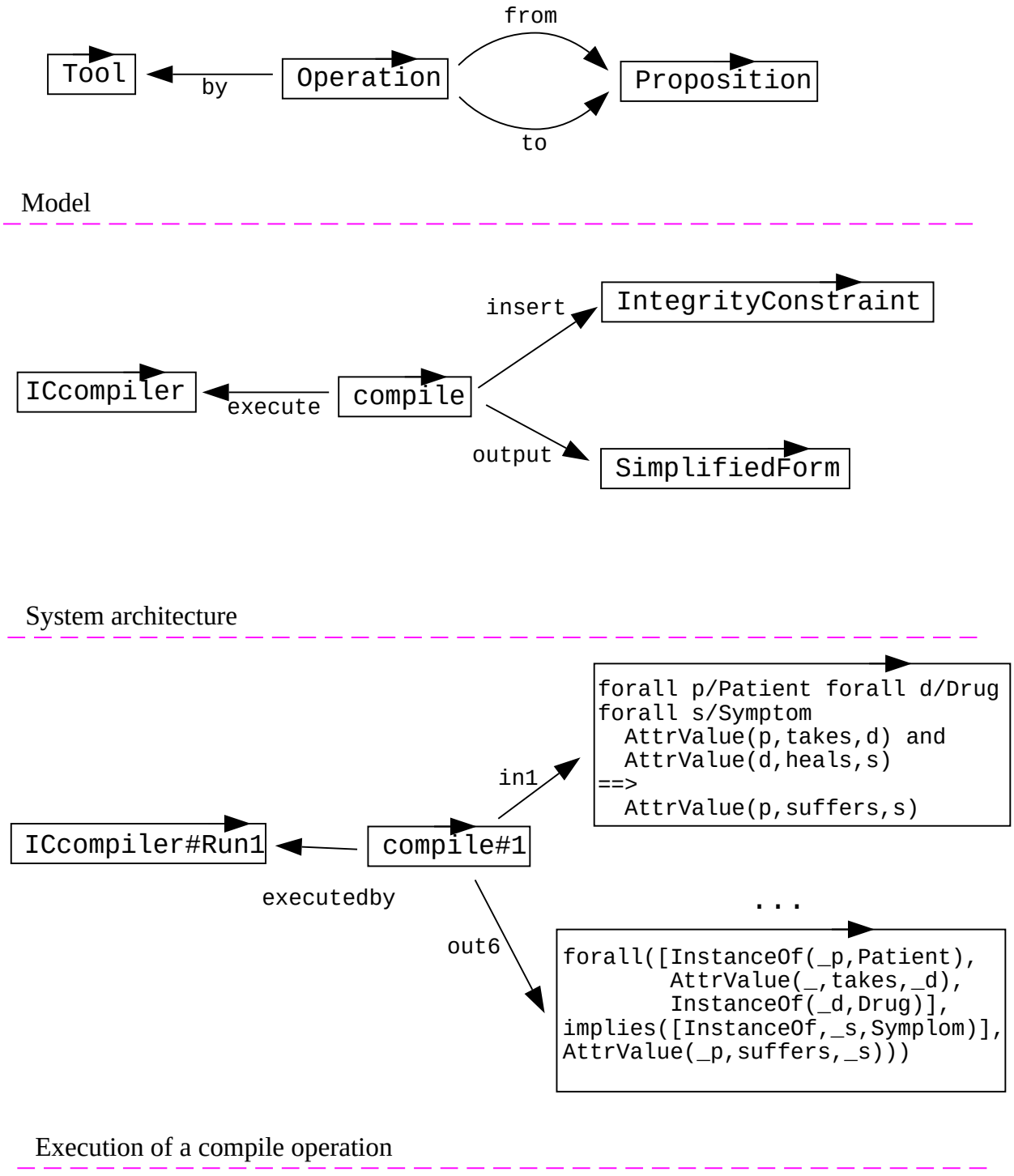

Fig. 4: Model of tools, operations and propositions and its application to the compilation of constraints

whenever a proposition, say x, is inserted or deleted. ConceptBase automatically generates the messages by searching the operations and tools that

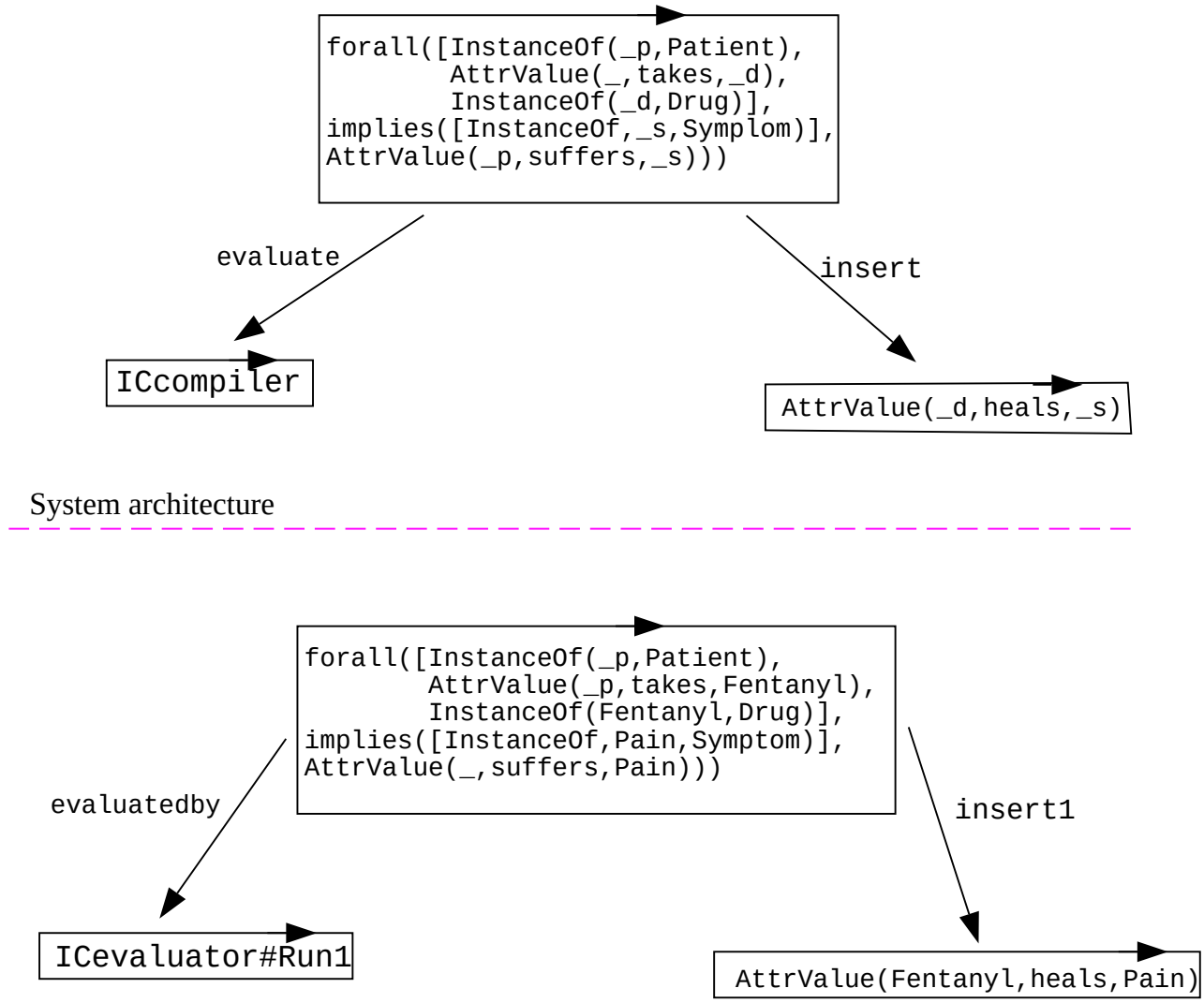

Fig. 5: Application of the model to constraint evaluation

are connected to the classes of x. For the two examples of figures 4 and 5 we have the following messages:

```
ICcompiler: execute compile
                    insert=$forall p/Patient forall d/Drug ...$

ICevaluator: evaluate forall([InstanceOf(_p,Patient),...)
```

```
insert=$AttrValue(Fentanyl,heals,Pain)$
```

Representing system behavior as part of a consistent KB has the benefit that this information is also checked for integrity. An important example are the concerned classes of section 3. ConceptBase stores them as propositions of the form `Proposition(`*oid*`,`*lit*`,*dependson,`*c*`)`. Referential integrity prevents deletion of the concerned class *c* for the lifetime of the literal occurrence *lit*. In the previous example, the literal occurrence `AttrValue(p,takes,d)` is connected to the `takes` attribute of `Patient` (see fig. 1) by

```
Proposition(#3591.127,$AttrValue(p,takes,d)$,*dependson,#3591.20)
```

If such a concerned class for a literal cannot be found at the compile time of a formula, this formula refers to a class not existing in the KB and is rejected. For the same reason, the declarative specification of a predicative formula may not be removed as long as its simplified forms are part of the system. That means: the relationship between specification and implementation is maintained automatically (see also [JJR89]).

On the other hand, using the tool model to describe the system architecture is itself a kind of system specification. Just the insertion or deletion of an operation and its links to propositions and tools will alter the behavior of the system w.r.t. availability or non-availability of the operation. Within the Esprit project DAIDA [JMSV90] a version of the tool model specialized for software process modeling was used to interconnect distributed environments for database software specification, design and implementation by the concept of a global knowledge base. Representation of such tools as part of the application knowledge base promises that the notion of database integrity can be re-used to describe consistent environments.

5. Conclusions

In this paper we presented a method for checking integrity and interpreted it for object-oriented systems. As a test case we implemented the ideas within the system ConceptBase. We see the following contributions:

1) A deductive integrity checking method is proposed with a formula compilation phase independent from the database state and an evaluation phase checking integrity of transactions by using code generated in the first phase. Only rules affecting constraints are regarded for integrity checking.

2) Simplification of formulas is supported down to the level of attribute updates as opposed to algorithms in the relational domain where updates occur on complete tuples.

3) The method treats the case where constraints (or rules) are updated, i.e. inserted or deleted, at run-time of the system. The incremental compiler of ConceptBase uses explicit links between the declarative specification and the executable code to maintain the system.

4) The update which instantiates a simplified form is linked to classes. Thus, exploitation of search space limiting data structuring, in particular specialization hierachies, is supported.

5) Rules and constraints are introduced to object-oriented data models by stating dependencies between the literals and class objects. Behavioral aspects such as compilation and evaluation are stored as first class citizens in the KB. Active databases [DBM88] and some object-oriented databases (e.g. [RRR*88]) do the same but rely on hand-written procedures rather than on declarative specifications. Additionally, in our approach the dependencies and behavioral aspects are subject to consistency themselves.

Several important problems and opportunities remain to be explored. Firstly, the management of redundant derived information at both the object level (data derived by rules) and the metalevel (evaluators derived by compilation) seems to offer significant promise for further efficiency gains [JJM90]. Secondly, we plan to extend the compilation techniques used here to the case of complex configured objects as studied in [GOCE90].

The current strategy of resolving an integrity violation is simply to reject the transaction. A more sophisticated approach that provides automatic "repair" procedures generated from integrity constraints is presented in [CW90]. Moreover, there is no provision to exploit the interaction of multiple updates [HI85,SV87] for further formula simplification and restriction of the concerned classes. If one considers modifications not as a pair of insert and delete updates within a transaction but as a single operation, one could

also cut off certain integrity constraints, e.g. those demanding the existence of a modified attribute.

Finally, future work concerns the integration of predicative and temporal constraints in Telos. The latter can be used to specify the behavior of the system by relating the time of a tool invocation to the system time of a new object.

6. References

[ABD*89] Atkinson,M., Bancilhon,F., DeWitt,D., Dittrich,K., Maier,D., Zdonik,S. (1989). The object-oriented database system manifesto. *Proc. First Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan.

[ADV90] Committee for Advanced DBMS Function (1990). Third-generation database system manifesto. Memorandum UCB/ERL M90/28, University of California.

[BL86] Brachman,R.J., Levesque,H.J. (1986). What makes a knowledge base knowledgable? A view of databases from the knowledge level. *Proc. First Int. Conf. on Expert Database Systems*, Menlo Park, Calif.

[BDM88] Bry,F., Decker,H., Manthey,R. (1988). A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. *Proc. EDBT'88*, Venice.

[BMM90] Bry,F., Manthey,R., Martens,B. (1990). Integrity verification in knowledge bases. ESPRIT BRA Compulog No. 3012, ECRC deliverable No. D.2.1a.

[CW90] Ceri,S., Widom,J. (1990). Deriving production rules for constraint maintenance. *Proc. 16th Int. Conf. on Very Large Databases*, Brisbane, Australia.

[DBM88] Dayal,U., Buchmann,A., McCarthy,D. (1988). Rules are objects too: A knowledge model for an active object-oriented database system. *Proc. 2nd Int. Workshop on Object-Oriented Database Systems*, Bad Münster, Germany, 129-143.

[DEC86] Decker,H. (1986). Integrity enforcement on deductive databases. *Proc. First Int. Conf. on Expert Database Systems*, Menlo Park, Calif.

[EJJ*89] Eherer,S., Jarke,M., Jeusfeld,M. Miethsam,A., Rose,T. (1989). ConceptBase V2.0 user manual. Report MIP-8936, Universität Passau, Germany.

[GOCE90] Gocek,M. (1990). Entwurf und Realisierung eines interaktiven Assistenten für konzeptuelle Versions- und Konfigurationsentscheidungen. Diploma thesis, Universität Passau, Germany.

[HI85] Hsu,A., Imielinski,T. (1985). Integrity checking for multiple updates. *Proc. ACM-SIGMOD Int. Conf. on Management of Data.*

[JJM90] Jarke,M., Jeusfeld,M., Miethsam,A. (1990). Redundancy in integrity management for deductive object bases: a preliminary investigation. ESPRIT BRA Compulog No. 3012, Deliverable No. D.2.1b.

[JJR89] Jarke,M., Jeusfeld,M., Rose,T. (1989). Software process modeling as a strategy for KBMS implementation. *Proc. First Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan.

[JMSV90] Jarke,M., Mylopoulos,J., Schmidt,J.W., Vassiliou,Y. (1990). Information systems development as knowledge engineering: a review of the DAIDA project. To appear in *Programirovanie*, Soviet Union, also in: Report MIP-9010, Universität Passau, Germany.

[KG90] Kießling,W., Güntzer,U. (1990). Deduktive Datenbanksysteme auf dem Weg zur Praxis. *Informatik Forschung und Entwicklung* 5(1990), Springer Verlag.

[KMSB89] Koubarakis,M., Mylopoulos,J., Stanley, M., Borgida,A. (1989). Telos: features and formalization. Technical Report KR-89-04, University of Toronto, Ont.

[KS88] Kowalski,R., Sadri,F. (1988). A theorem-proving approach to database integrity. In Minker,J. (ed.): Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers.

[KRÜ89] Krüger,E. (1989). Integritätsprüfung in deduktiven Objektbanken am Beispiel von ConceptBase. Diploma thesis, Universität Passau, Germany.

[LV89] Lefebre,A., Vieille,L. (1989). On Deductive Query Evaluation in the DedGin* System. *Proc. First Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan.

[LST86] Lloyd,J.W., Sonnenberg,E.A., Topor,R.W. (1986). Integrity constraint checking in stratified databases. Technical Report 86/5, Department of Computer Science, University of Melbourne.

[LHW90] Lyngbæk,P., Hasan,W., Wilkinson,K. (1990). The IRIS kernel architecture. *Proc. EDBT'90*, Venice.

[MAN90] Manola,F. (1989). An evaluation of object-oriented DBMS developments. Technical Report TR-0066-10-89-165, GTE Laboratories, Waltham, MA.

[MBJK90] Mylopoulos,J., Borgida,A., Jarke,M., Koubarakis,M. (1990). Telos: a language for representing knowledge about information systems. *ACM Trans. Information Systems*, to appear

[NT89] Naqvi,S., Tsur,S. (1989). A logical language for data and knowledge bases. Computer Science Press.

[NIC79] Nicolas,J.-M. (1979). Logic for improving integrity checking in relational databases. Technical Report, ONERA-CERT, Toulouse, France, also in: *Acta Informatika* 18(3), Dec. 1982.

[RRR*88] Rehm,S., Raupp,T., Ranft,M., Längle,R., Härtig,M., Gotthard,W., Dittrich,K., Abramowitz,K. (1988). Support for design processes in a structurally object-oriented database system. *Proc. 2nd Int. Workshop on Object-Oriented Database Systems*, Bad Münster, Germany.

[SV87] Simon,E., Valduriez,P. (1987). Design and analysis of a relational integrity subsystem. MCC Technical Report No. DB-015-87.

[STAU90] Staudt,M. (1990). Anfragerepräsentation und -auswertung in deduktiven Objektbanken. Diploma thesis, Universität Passau, Germany.

[SJGP90] Stonebraker,M., Jhingran,A., Goh,J., Potiamos,S. (1990). On rules, procedures, caching and views in database systems. Memorandum UCB/ERL M90/36, University of California.