Accepted for GI-Conf. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), Dresden (Germany), March 1994

Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract

Andreas Geppert, Klaus R. Dittrich

Institut fuer Informatik, Universitaet Zuerich, Switzerland
{geppert | dittrich}@ifi.unizh.ch

Abstract

We describe an approach to the specification and implementation of consistency constraints in object-oriented database systems, adopting the programming-by-contract paradigm developed for object-oriented programming. We also investigate how consistency constraints specified in programming-by-contract can be transformed into production rules of an active, object-oriented database system.

Keywords: object-oriented database systems, consistency constraints, active database systems

1 Introduction

In addition to the tasks of modeling, storing, and retrieving data, a database system (DBS) has to prevent database states that do not represent legitimate models of the miniworld of interest. The DBS has to perform *consistency maintenance*, which comprises three tasks:

- the DBS has to provide means to *define* consistency constraints pertaining to the miniworld under consideration,
- inconsistent situations in the database have to be *detected*, and
- in case of inconsistencies, consistency has to be *enforced*.

We investigate these three tasks in the context of *object-oriented database systems* (ooDBSs). In this context, the requirements for consistency maintenance are as follows:

- consistency constraints should be defined in a declarative way instead of as code fragments,
- the approach to define consistency constraints should be well integrated with the object-oriented style of defining and manipulating classes and objects,
- the approach to define consistency constraints should take the behavior of objects into account,
- constraint maintenance should be applicable to persistent and volatile (non-persistent) objects (provided that applications access and manipulate both persistent and non-persistent objects),
- detection of consistency violations should be efficient,
- the concept should support efficient repairing of inconsistent situations.

In this paper, we adopt the Programming-by-Contract paradigm (PbC) [15] developed for object-oriented programming languages for consistency maintenance in object-oriented database systems. This approach will support implicit, dynamic (two-state transition) and a restricted form of behavioral consistency constraints.

We investigate the implementation of constraint violation detection and consistency enforcement on top of the active object-oriented database management system (aDBMS) SAMOS [8]. Since SAMOS is intended as a general platform for various functionalities that benefit from an active mechanism, it is also an interesting issue how well PbC can be implemented on top of SAMOS, or where potential limitations may be encountered.

The remainder of this paper is structured as follows. The next section describes the foundations for the paper: consistency constraints in database systems and the programming-by-contract principle. This section also introduces a running example. Section 3 describes how consistency constraints in an ooDBS are specified using the PbC paradigm. Section 4 shows how the specified consistency constraints are implemented through an active mechanism. Section 5 surveys related work, and section 6 concludes the paper. In Appendix A, we shortly investigate how our approach of consistency constraints checked at runtime can be combined with compile time transaction proofs.

2 Consistency Constraints in Database Systems and the Programming-by-Contract Paradigm

2.1 Consistency Constraints in Database Systems

The purpose of consistency constraints (CCs) is to restrict the database states (or sequences thereof) to those that are considered as legal. Therefore, constraints as present in the modeled miniworld have to be expressed in a formal way, i.e., must be made known to the DBS.

In general, consistency constraints can be classified according to where they are specified into *internal* and *external* consistency constraints.

Internal CCs are those that are known to and can be enforced by the DBS, while external CCs have to be expressed, checked, and enforced within application programs. Internal CCs can be subdivided into the following three types [7]:

- *inherent* consistency constraints: they are fixed for a given data model and therefore do not have to be specified (e.g., absence of cycles in inheritance relationships),
- *implicit* consistency constraints: they can be specified in the schema using specific constructs (keywords) of the data definition language (DDL), e.g., the identification of *unique* properties,
- *explicit* consistency constraints: they are formulated in a separate sublanguage of the DDL, the constraint definition language, either as predicates in a declarative language or by means of triggers. As an example, consider the typical consistency constraint that employees may not earn more than their manager.

Another classification of consistency constraints is based on whether only single database states or even database state transitions can be constrained [7]:

- *static* consistency constraints consider only single states of a database,
- *dynamic* CCs allow to constrain state transitions, i.e., they specify which transformations of database states into new ones are allowed. In the general case, dynamic CCs constrain arbitrary sequences of state transitions. Two-state transitions are a special case.

Finally, the third relevant classification of consistency constraints determines which properties of entities can be subject to constraints:

- *state* constraints specify conditions on the values (states) of entities,
- *behavioral* constraints further allow to constrain the behaviour of entities, e.g., specify correct semantics of methods.

Note that the last two classification dimensions are orthogonal to each other.

2.2 Programming by Contract

PbC [15, 16] is an approach to achieve correct and robust object-oriented, modular programs. In addition to the usual parts of class definitions, preconditions, postconditions, and class invariants are specified.

In PbC, preconditions define the conditions that must be fulfilled in order to execute a method correctly. Preconditions are formulas over the state of the receiver and the arguments of the message. Postconditions constrain the state of the receiver object after method execution as well as the results of the method. Postconditions are formulas over the final state, the initial state, and the output parameters. Note that in postconditions the old state of the receiver can be referred to, although the initial state might be modified by a method. Syntactically, the old value of an instance variable can be referred to through the keyword **old**. Together, preconditions and postconditions define a *contract*:

- the precondition defines the conditions the sender of a message has to obey,
- the postcondition defines what the receiver (the server) is obliged to produce.

Constraints that are not specific for a method but restrict the permissible states of any object of a class can be defined through *invariants*. An invariant is attached to its class and is guaranteed to hold at the end of each method execution for any instance of the class.

2.3 A Running Example

We consider the following mini world as a running example. The schema models a reuse-based CASE-environment. Modules are components that can be composed out of other components. Each module has as attributes an interface, a design, and an implementation. For designs, there can be different alternative design choices. DesignChoice in turn is a set of alternative designs . Designs of different designChoices can be related through the relationship incompatibilities , i.e., may not be used together within the same construction. Designs are realized through implementations . Implementations can be verified and tested. Each kind of artifact requires a documentation .

Example 1 shows the relevant class definitions. We restrict the schema to the parts necessary for subsequent examples (see also Figure 1 for the relevant classes).

```
class SWArtifact
        string      name;
        Person      author;
        Date        date;
        Docu        documentation;

class Module:SWArtifact
        Interface       interface;
```

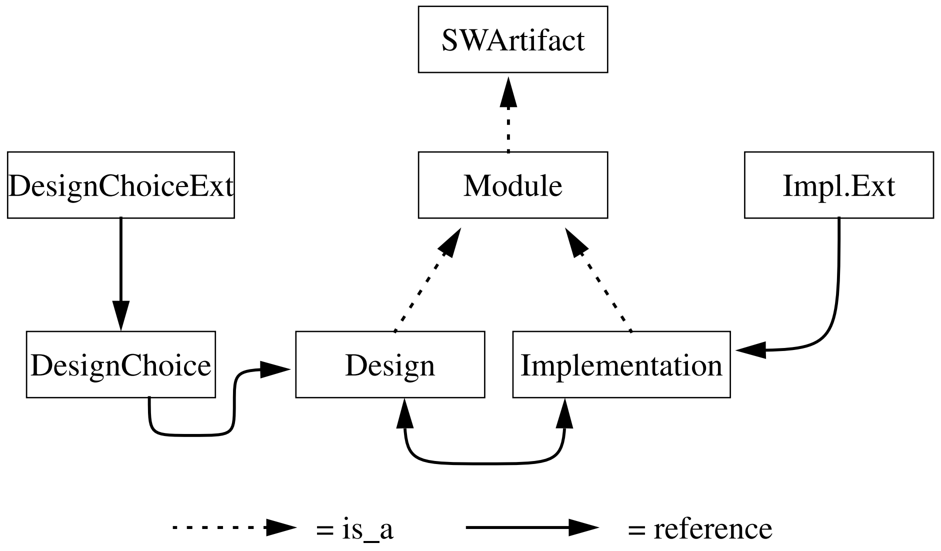

Figure 1. Sample Schema

```
        Design              design;
        Implementation      impl;                          // the current impl. of design
        set <Module>        components;
        int                 state
        method addComp (Component newComp);

class Design:Module
        set<Implementation>     impl;                      // all possible implementations
        set<Design>             incompatibilities;
        Text                    designDesc;
        method addImpl (Implementation I);

class Implementation:Module
        Text                code;
        set<Module>         usedIn;

class DesignChoice
        set<Design>     choices

class DesignChoiceExt                                      // extension class for DesignExt
        set<DesignChoice>       elements;

class ImplementationExt                                    // extension class for Implementation
        set<Implementation>     elements;
        method addImpl (Implementation I);
        method removeImpl (Implementation I);
```

Example 1. Schema for Sample Class Definitions

In this context, we may define the following constraints:

CC1: Alternatives of different design choices may only be used within the same module if the alternatives are not related via the relationship incompatibilities .

CC2: Possible implementations for a design must be tested.

CC3: If a component is to be added to a module, the new component must have a documentation.

CC4: For each design choice, there have to be at least two alternatives (otherwise, there is no choice).

CC5: Implementations can only be removed from the extension of implementation if they are not used by any module .

CC6: Modules can only be approved by the manager of the author and only if the module is in the state tested .

3 Specification of Consistency Constraints

In this section, we introduce the assumptions we make for the underlying data model. We then introduce *Programming by Contract for Databases* (PbC-DB) and describe how consistency constraints can be specified using this paradigm.

3.1 The Object Model

We require an object model as provided by most current ooDBSs [2]. In particular, we assume that the definition of *classes* comprises a unique name, a type, a list of supertypes, and a set of method signatures. Each *method signature* defines a method name, a list of typed formal parameters, and possibly a return value. The database schema then consists of a set of class definitions.

Furthermore, the object model supports the concept of *extension*. An extension is a collection of *currently existing instances* of a specific class (or of subclasses thereof), while the class definition only specifies the *intension* of the class. The structure and behavior of extensions are in turn defined through classes.

We further consider database instances as collections of extensions. Each database has a unique schema assigned. We finally assume that the DBS also supports the definition and implementation of *application programs*. Throughout this paper, we require that method and application implementations obey the encapsulation of objects.

3.2 Requirements and Solution

In addition to the aforementioned requirements, additional ones are obtained when considering the granularity of CCs. It is apparently not sufficient to consider consistency just local to objects. CCs may also concern multiple, related objects (*inter-object CCs*). Moreover, it is not sufficient to just consider objects that are consistent one by one: each extension of an object class may require additional specific CCs. Finally, even if extensions and single instances are consistent, it may be the case that an application implements its purpose correctly only if specific conditions are valid. In particular, we want the database system to control consistency of entire databases and application programs (which is not possible, e.g., in relational systems). Hence, we obtain as a first requirement that consistency can be defined on the appropriate level: for objects, extensions, and entire databases/applications. In doing so, CCs can be kept local to the data they belong to.

Secondly, in order to provide for a "user-friendly" constraint definition language, CCs for objects, extensions, and applications must be specified in the same language.

Subsequently, we will define consistency maintenance mechanisms on three levels: the object level, the extension level, and the database level. Although these three levels are different from a semantic point of view, it turns out that constraints on all levels can be specified using the same language.

3.3 Constraints for Objects

3.3.1 Specification of Constraints

In PbC-DB, we express consistency constraints through invariants, pre- and postconditions.

Invariants are constraints that have to hold for all instances of a class. For classes, this means that their definition contains a further clause for the invariant namely:

invariant name formula [**repair** action] { **,** name formula [**repair** action] } **;**[1]

Formulas are written as conjunctive forms. Each conjunct is a disjunction of predicates. Predicates can also contain range-restricted, quantified predicates. The only free variables permitted in formulas are the instance variables of the class. Methods are also permitted in formulas, as long as they do not perform modifications on the state of objects (i.e., only observers are allowed).

In formulas, it is also possible to refer to methods and instance variables of other objects of other classes if these objects are referenced. That is, we permit to violate encapsulation in constraint definitions[2]. Such *inter-object constraints* are necessary whenever a (real world) condition is not completely local to one object, but the consistency of one object depends also on the properties of other objects. Inter-object constraints require that the objects related by a constraint reference each other (i.e., inverse references must be defined).

The **repair** clause specifies an action to be performed whenever the invariant has been recognized to be violated. Actions are code fragments in the object model's data manipulation language. The user is responsible to specify only those repairs that lead to a consistent object state. Inconsistent object states resulting from a repair will be detected and will lead to further repairs, thus non-terminating repair cycles may happen in such a case.

Note that unlike in plain PbC, invariants are named here. An invariant can be equivalently written as several formulas (each with a distinct name), or as a large conjunction. Although this equivalence holds, it may be better in some cases to specify several clauses, especially if different repair actions are related to the formulas. Another reason for writing multiple invariants refers to the checking of invariants, to be discussed below.

---

invariant compat // invariant in class Module
  forall components c, components d: ! (c.design in d.design.incompatibilities)

Example 2. Sample Invariant for class Module (CC1)

---

1. In syntax definitions, we use the following conventions: keywords are written in bold letters. A phrase enclosed in square brackets is optional, and a phrase in set brackets can be repeated arbitrarily many times (including zero). "!" stands for negation and "&&" for conjunction.
2. In this way, the schema designer is not forced to extend class interfaces only for the sake of CC definitions.

CCs that define a contract (i.e., requirements concerning the parameters of a method, semantics of method execution, and object states at the end of a method execution) are specified through pre- and postconditions. Similar to class definitions, method signatures are extended as well. In addition to the initial definition of methods, pre- and postconditions become a part of method definitions. They have a similar syntax to invariants:

require name formula [**repair** action] { **,** name formula [**repair** action] } **;** // preconditions
ensure name formula [**repair** action] { **,** name formula [**repair** action] } **;** // postconditions

---

require testedImpl // precondition for method addImpl in Design
(CC2)
I.state == TESTED

ensure docExists // postcondition for addComp in class Module (CC3)
newComp.documentation != NULL

Example 3. Sample Pre- and Postconditions for Methods

3.3.2 Semantics of Constraints

The semantics of constraints assigned to classes or methods are as follows.

Whenever a message *m* is sent to an object *o*, the precondition of the method named *m* assigned to the class of *o* is checked (see below how this check is performed). If none of the formulas in the precondition evaluates to false, the method body is executed. Suppose however that a set of formulas of the precondition evaluate to false; if any of them has no repair action defined, the method execution is aborted. In general, we assume that each method body is implemented as a transaction, so aborting the method means to abort the corresponding transaction. If all formulas evaluating to false have repairs defined, these repairs are executed after the condition check.

An alternative approach is taken in ODE [10], which allows two checking phases: after repairs have been executed, the constraints are checked again. If then one condition is still violated, the method is aborted. In our approach, however, constraint violations at the end of repairs lead to nested execution of repairs. Similarly, Starburst [4] allows multiple iterations, at the price of considering termination of multiple rules, whose repair actions may violate each others formula.

The same procedure for constraint checking and enforcement applies to postconditions and invariants. They are checked at the end of the method execution after the last statement of the method body (but before control leaves the method execution).

Invariants for classes and conditions for methods lead to the notion of *object consistency* (or class consistency). The specification and enforcement of these constraints results in legitimate states of objects and method executions (whereby the notion of legitimacy is user-defined). Still, constraints pertaining to the extensions or applications might be violated, although each object in the universe is consistent in itself. Therefore, the next two subsections describe how constraints can be specified for extensions and applications.

3.4 Constraints for Extensions

Constraints as defined above only refer to single instances of classes. However, it will usually also be necessary to constrain *extensions* of instances for a given class (and its subclasses).

Therefore, we apply the PbC-DB paradigm as outlined above to extensions, too. Recall that we assume that extensions are collection-valued objects and that they are defined by own classes. The role of the various constraints is the following:

- the constraints pertaining to the extensions in their entirety (instead of each single object) are formulated as part of the invariant of the extension,
- the constraints that an object and the extension have to obey in order to make the instance a member of the extension are expressed as preconditions of insertion methods,
- the circumstances under which an object can be legally removed from the extension are specified as postconditions of deletion methods.

As an example, consider the extension of DesignChoice (DesignChoiceExt). Only design choices that have at least two alternatives can be added to the extension. This is a precondition of the method addChoice of the extension class. Secondly, only implementations that have not been used in any module may be removed from the set of existing implementations: this is the postcondition of the method removeImpl of the extension class ImplementationExt.

```
require AlternativeChoices                    // precondition for method addChoice (CC4)
    count (newChoice.choices) >= 2

ensure ImplNotUsed                            // postcondition for method removeImpl (CC5)
    oldImpl.usedIn == NULL;
```

Example 4. Constraints for Extensions

Apparently, constraints for extensions are specified in the same way as constraints for objects/classes. They are also checked in the same way as described above. This leads to a second "level" of consistency: *extension consistency.* This property means that each extension in the database respects its invariants, and each method defined for an extension class obeys its pre- and postconditions. Note that extension consistency subsumes object consistency, since only consistent objects can be members of extensions.

3.5 Constraints for Databases and Applications

Extension and object consistency do not yet ensure that application programs behave in a consistent manner. Therefore, constraints for applications are required as well. To do so, we consider a database as a complex object and the applications operating on it as an equivalent to the methods of a class[3]. In other words, the "database object" has the extensions, possible other persistent variables, and the application programs as its properties.

Note that in this way the database system controls the consistency of applications, in addition to the consistency of database states. The types of constraints play the following role for databases and applications:

3. This is similar to the notions of "application" and "application program" in O_2 [7].

- constraints on the database states that are not already specified for extensions or classes can be formulated as invariants of the database object,
- constraints that have to hold in order to correctly execute application programs are specified as the precondition of the application,
- constraints that have to hold at the end of a program execution are defined in the postcondition of the application.

In Example 5, we consider an application program whose semantics are to approve a module mod and whose postcondition is given in CC6.

```
ensure  Approval                                        // postcondition of application approve
   old  mod.state == TESTED && approvingMgr == mod.author.manager &&
   mod.state == APPROVED
```

Example 5. Constraints for Applications

Consistency constraints for databases/applications are checked in the same way as those for extensions or objects (i.e., at the begin and end of application transactions). This leads to the third level of consistency: *application consistency*. Note that application consistency subsumes extension consistency (and therefore also object consistency).

4 Implementation of Consistency Constraints and Consistency Enforcement

This section investigates how consistency maintenance in PbC-DB can be implemented. Similar to other authors [e.g., 4], we follow the approach to use an *active DBMS*. An aDBMS is able to detect events (and situations) in the database and beyond and to react accordingly. For consistency maintenance, the situations to be detected are the potential consistency violations.

We use the aDBMS SAMOS [8] for implementing PbC-DB. For the sake of comprehensiveness, we give a short survey of the active mechanism of SAMOS before we describe the transformation process of constraints into *production rules* (ECA-rules).

4.1 The Active Database System SAMOS

SAMOS [8] supports an active mechanism in addition to the usual features of an object-oriented database system. Active behavior is specified in terms of production rules. Each production rule consists of an event, a condition, and an action part. Events can be *primitive* (method events, value events, time events, transaction events) or *complex* (sequence, disjunction, negation and so forth) [9]. Method events can refer to the point of time before the method execution starts (expressed through the keyword **BEFORE**), or to the point in time after the method execution (**AFTER**). Transaction programs can be named, and the name can be referred to in transaction events. Such an event is then signalled only when an execution of the corresponding "transaction type" issues the transaction command.

The condition is a function expressed in the DML of the underlying ooDBS that returns a boolean value. Actions are also code fragments written in the DML.

SAMOS distinguishes between *class-internal* and *-external* rules. Class internal rules are part of the class definition, they can directly read or modify the instance variables of instances of the class they are defined for. They can only be defined by the class-implementor, not by any arbitrary user. Class-external rules have to respect encapsulation, and are therefore not permitted to read or modify instance variables directly.

The execution model is realized on top of the object-oriented database system ObjectStore [13] and uses the nested transaction model [17] of ObjectStore. The coupling mode for each rule defines when the rule is executed with respect to the triggering event: immediately (directly after the triggering event has been detected), deferred (before the triggering transaction commits), or decoupled (in a separate transaction). If multiple rules are defined for the same event, the execution order of the corresponding rules can be specified through priorities.

SAMOS supports nested rule execution: during the execution of a rule, new events may occur and trigger the execution of further rules.

4.2 Implementing Constraints through Production Rules

The General Translation Process

In principle, each constraint definition is translated into a production rule definition. All the generated production rules are class-internal. Furthermore, we assume that each method that performs modifications on the state of its receiver and that has constraints attached is realized as a transaction (i.e., as a subtransaction of the sender of the corresponding message).
For preconditions, the event (of the resulting production rule) is defined as a sequence of a message sending event and the begin of transaction (BOT) of the method body. In order to signal the BOT of the right transaction, the transaction event has as parameter the name of the transaction implementing the method (the concatenation of the class and the method name). Note that the message sending alone would not be sufficient as event specification. In this case, the transaction executing the method body has not yet started, and an abort would abort the transaction of the message sender. However, the message sending event is necessary in order to determine the "right" receiver of the message.

The condition is translated into a DML-fragment that computes the negated truth value of the constraint. The action is either the repair function, if one has been defined, or the abort command. The coupling mode is immediate.

The same translation applies to postconditions, with the exception that the rule has to be triggered after the execution of the method body (indicated through the transaction event **EOT**). Similar to the case of preconditions, message events (i.e., the SAMOS message event **AFTER**) are not appropriate, since the transaction can then already be committed, and an abort would rollback the transaction of the message sender. Therefore, events for production rules implementing postconditions are implemented as a sequence of a message event and a transaction event, too.

Production rules realizing invariants are triggered upon value events (i.e., any modification of some instance variable). Condition and action parts are determined in analogy to the case of pre- and postconditions. The coupling mode for invariants is deferred, as invariants may be violated during a method execution but have to hold at the end of the method execution.

Table 1 summarizes the translation rules. c denotes the name of a class for which a constraint is defined, and m a method for which pre- or postconditions are specified.

Kind of Constraint	Event	Condition	Action	Coupling Mode
precondition	BEFORE.c.m; BOT c_m	! (pre)	repair or abort	immediate
postcondition	BEFORE c.m; EOT c_m	! (post)	repair or abort	immediate
invariant	UPDATE (iv)	! (invariant)	repair or abort	deferred

Table 1. The Translation of Constraints into SAMOS Production Rules

```
ON    BEFORE.removeImpl ; BOT ImplementationExt_removeImpl
IF    ! (oldImpl.usedIn == NULL)
DO    abort;
```

Example 6. Generated ECA-Rule for Postcondition CC5

Generation of ECA-Rules for Inter-Object Constraints

Events as defined above are sufficient for intra-object constraints. However, inter-object constraints (invariants) defined for the class of one object can be violated by methods executed for another object. Hence, whenever an invariant refers to other objects, the event definition of the generated ECA-rule has to reflect this possibility. Concretely, let an invariant I of class C refer to the related class C'. The compiler then generates an ECA-rule that triggers upon the modification of states of a C'-object. The condition checks whether the invariant I is violated, and the action eventually executes I's repair.

ECA-Rules for State-Transition Constraints

In PbC-DB, postconditions can refer to the state of the receiver as of the beginning of the method execution. Some systems such as Starburst or NAOS [6] support deltas (i.e., old and current states of objects), which are not provided by SAMOS. Therefore, old states have to be recorded by the generated *implementation* of condition evaluation functions. During the compilation of a postcondition, it is checked whether the postcondition refers to the old state. If yes, the object is copied at the beginning of the method execution, and this copy is supplied to the condition function as a parameter.

Optimization

Given the above translation, SAMOS would possibly execute the same rule multiple times for invariants. In fact, SAMOS would trigger one production rule for each modification of the instance variable iv within a method execution. A better solution using the event definition facilities of SAMOS [9] specifies that each invariant has to be checked only once during each method execution. Concretely, let iv_1 through iv_n be the instance variables referred to in one invariant constraint. The event of the production rule is then defined as

$$*(\text{ UPDATE } (iv_1) \mid ... \mid \text{UPDATE}(iv_n)\ ) : \text{same transaction}$$

The bars (|) define a disjunctive event. This disjunction specifies that the invariant production rule has to be triggered whenever one of the instance variables is modified. The asterisk (*) together with same transaction specifies that only the first occurrence of the disjunction within a method execution (read: transaction) is signalled, while each subsequent occurrrence is “forgotten”. Figure 2 shows a sample method execution that contains three updates of two instance variables `a` and `b` (and therefore signals three value events). Using the *-operator, only the first event triggers the corresponding ECA-rule.

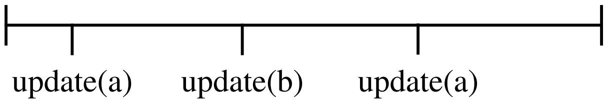

Figure 2. Sample Method Execution with Value Events

Termination

In general, it may occur that the execution of ECA-rules for constraint enforcement does not terminate. This is the case whenever the actions of one or more rules cyclically satisfy each others condition (i.e., the repairs violate each others constraint). This is a general problem of systems providing nested rule execution [e.g., 3, 4]. Although we would prefer a compile-time analysis of ECA-rules (which would then, e.g., determine whether termination is guaranteed), such an analysis tool is not yet available for SAMOS. It is therefore in the responsibility of the rule specifier that he/she ensures that no repair action violates any other constraint (i.e., it is a true repair). In this case, non-terminating rule execution during constraint enforcement cannot occur[4].

The current implementation of SAMOS allows to specify a maximum for the nesting depth of transaction trees. If this maximum is reached, SAMOS does not allow further subtransactions to begin execution. Albeit termination can be enforced in this way, it is still a brute-force solution, since once the nesting limit is reached no further rules can be executed (e.g., repairs).

4.3 Discussion

The above elaborations have shown that a powerful consistency constraint definition language such as PbC-DB can be implemented on top of a general-purpose active database system. The corresponding transformation benefits from powerful event definition facilities (value events, method events, transaction events, the *-operator, the *same transaction* keyword).

In general, the same functionality can be achieved by modifying the programming language and the language runtime system. The constraint maintenance mechanism is then a concept built into the language, for example as in Eiffel. However, using an aDBMS does not require to modify the programming language. Additionally, our approach can be used from different programming languages and query languages (which is not easily possible in the case of a tight integration with the programming language). Furthermore, since the active mechanism can also be used for further purposes, implementing constraints through ECA-rules contributes to the minimality of concepts needed in the DBMS [12].

Nevertheless, we have also recognized some possible deficiencies while developing the transformation process. First, an aDBMS useful for implementing (even simple) dynamic constraints should provide for deltas, while in SAMOS they must be implemented by the consistency constraint compiler. Second, it is still open how to determine termination and other

4. In the O_2 prototype, rules are deactivated during rule execution in order to prevent cycles [3]. Since the effects of rule execution are much harder to comprehend in this way, we do not follow this approach.

properties of ECA-rule executions in object-oriented aDBSs. If rule execution (say, for constraint maintenance) is not tailored to the needs of the application purpose (such as in ODE [10]), then a general-purpose active object-oriented database system needs tools to analyze sets of rules.

5 Related Work

The first approach that investigates a declarative language for the specification of constraints and production rules for implementation has been developed for Starburst [4]. A constraint specification defines a table list (the relations that are affected by the constraint) and an SQL predicate. Since the only possible events in Starburst are value events (in the SAMOS terminology), the set of possible *invalidating operations* has to be determined (the set of operations each of which potentially violates a constraint). Constraints are implemented through production rules, where invalidating operations are the triggering events. Special attention is needed to keep the set of invalidating operations (triggering events) small in order to trigger rules only if necessary. Repairs can be defined by users.

A logic-based constraint definition language for non-standard applications and structurally ooDBS is proposed in [12]. Similar to Starburst [4], constraints are transformed into trigger definitions. In this work, constraints are not related to class or ADT definitions.

Another constraint maintenance mechanism has been proposed for the O_2 prototype [3]. In this approach, constraints can be specified as predicates. Repairs are written in the database programming language of O_2. Constraints are implemented by the (active) rule management subsystem of O_2. Predicates are mapped into queries of the O_2 query language. As in Starburst, triggering events (which are method events in O_2) have to be determined during this transformation. Since the rule mechanism does not provide for value events, invariants are hard to implement efficiently.

The active ooDBS Ode also supports the specification of constraints [10]. Similar to our approach, they attach constraints to classes. *Hard constraints* have the same meaning as postconditions in PbC-DB. Furthermore, our approach is similar to Ode in that both allow the specification of repairs. Another kind of constraints in Ode are *soft constraints*: they are not checked at the end of method executions, but at the end of the issuing transaction. Ode does neither support preconditions nor invariants, and also cannot monitor state transitions of objects.

The TM data model supports the specification of constraints on three levels [11]: the object level, the class level (extension in our terminology), and the database object level. Constraints are specified as logic formulas over the states of objects. TM assumes the ACID transaction model, i.e., constraints are checked at the end of transactions, and transactions are rolled back whenever they would produce an inconsistent state otherwise. Hence, the unit of consistency is a rather coarse one, and the notion of method is not very well integrated with the consistency mechanism.

Chimera is a database language that supports the declarative specification of constraints [5]. Constraints in Chimera are deductive, set-oriented rules; they can be defined for single or multiple classes. Similar to PbC-DB, a restricted set of transitions can be constrained through referring to **old** states. Constraints are translated into production rules. Similar to Starburst [4], events of production rules are the potentially invalidating operations, i.e., database update operations. It is unclear from [5] how constraints (pre- and postconditions) can be related to methods as in PbC-DB or Ode.

Finally, while all approaches surveyed so far allow only two-state transitions, [14] allows to constrain general state transitions. To do so, the underlying logic is enhanced with special operators for referring to states (i.e., the current state and the next state), and temporal quantifiers (e.g., always, sometimes).

Summarizing, while some of the concepts used in PbC-DB have been proposed elsewhere, it at least exceeds existing approaches in the combination of the provided features:

- it not only supports constraints on states of objects, but also on object behavior and state transitions, as far as they can be represented as two-state transitions.
- It is integrated well with the underlying object-oriented paradigm, since it fosters a class-specific notion of consistency. It is in principle seamlessly applicable for persistent and volatile objects.
- When compared to database-wide consistency constraints, PbC-DB contributes to modularity of persistent object-oriented systems (whereby modularity is a prerequisite for reusability of classes). The notion of locality of consistency constraints is reflected by the three levels of consistency.

6 Conclusions

In this paper, we have developed a consistency maintenance mechanism for object-oriented database systems (PbC-DB). This approach supports the declarative specification of constraints. It is integrated well with the object-oriented paradigm and in its original form has been proven successful for object-oriented programming. It allows not only to constrain states of objects, but also the behavior of objects and simple forms of state transitions. In general, the proposed constraint maintenance mechanism is more powerful than existing ones, since it allows for invariants, preconditions, and postconditions.

We have shown that a powerful constraint definition language such as PbC-DB can be implemented on top of an active object-oriented DBMS. In doing so, we have also learned some lessons about benefits and a few shortcomings of the used active mechanism.

Nevertheless, some questions remain open for future work. First, we would prefer to specify repairs in the same declarative style as constraints, or to propose possible useful repairs. For repairs that are more complex than simple transaction abort, the problem is apparently the understanding of classes and methods.

Second, PbC-DB allows for a natural view of class-specific and method specific notions of consistency. Hence, specifying consistency constraints should be an integral part of database design. A design methodology is needed that considers not only structures and behavior of objects, but also the consistency constraints pertaining to objects and classes.

Third and most important, we conceive it as a drawback that constraint specifiers are responsible for the termination of constraint enforcement. One part of our future work will therefore focus on defining “nice” properties of rule sets in object-oriented aDBSs and on developing methods how to analyze rules with respect to these properties (comparable to [1] for the relational case).

Acknowledgements

We gratefully acknowledge the comments of our colleagues Stella Gatziu, Stefan Scherrer, and Dimitris Tombros on an earlier version of this paper.

References

1. A. Aiken, J. Widom, J.M. Hellerstein: *Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism*. Proc. ACM-SIGMOD Intl. Conf. on Management of Data, San Diego, June , 1992.
2. M. Atkinson, F. Bancilhon, D.J. DeWitt, K.R. Dittrich, D. Maier, S.B. Zdonik: *The Object-Oriented Database System Manifesto (a Political Pamphlet)*. Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases, 1989.
3. C. Bauzer-Medeiros, P. Pfeffer: *Object Integrity Using Rules*. Proc. European Conference on Object-Oriented Programming, Geneva, Switzerland. LNCS 512, Springer 1991
4. S. Ceri, J. Widom: *Deriving Production Rules for Constraint Maintenance*. Proc. of the 16th Intl. Conf. on Very Large Data Bases, Brisbane, Australia, August 1990.
5. S. Ceri, P. Fraternali, S. Paraboschi: *Constraint Management in Chimera*. Bulletin of the IEEE Technical Committee on Data Engineering 17:2, June 1994.
6. C. Collet, T. Coupaye, T. Svensen: *NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System*. Proc. of the 20th Intl. Conf. on Very Large Data Bases, Santiago, Chile, September 1994.
7. O. Deux: *The O2 System*. Communications of the ACM 34:10, 1991.
8. R. Elmasri, S.B. Navathe: *Fundamentals of Database Systems*. Benjamin/Cummings Publishing, 1989.
9. S. Gatziu, A. Geppert, K.R. Dittrich: *Integrating Active Mechanisms into an Object-Oriented Database System*. Proc. of the 3rd Intl. Workshop on Database Programming Languages (DBPL), Nafplion, Greece, August 1991.
10. S. Gatziu, K.R. Dittrich: *Events in an Active Object-Oriented Database System*. In N.W. Paton, H.W. Williams (eds): Proc. Workshop on Rules in Database Systems, Edinburgh, UK, September 1993 (Workshops in Computing, Springer-Verlag, 1994).
11. N. Gehani, H.V. Jagadish: *Ode as an Active Database: Constraints and Triggers*. Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, Spain, September 1991.
12. P.W.P.J. Grefen, R.A. de By, P.M.G. Apers: *Integrity Control in Advanced Database Systems*. Bulletin IEEE Technical Committee on Data Engineering 17:2, June 1994.
13. A.M. Kotz: *Triggermechanismen in Datenbanksystemen*. IFB 201, Springer 1989.
14. C. Lamb, G. Landis, J. Orenstein, D. Weinreb: *The ObjectStore Database System*. Special Issue on Next-Generation Database Systems. CACM 34:10, 1991.
15. U.W. Lipeck, M. Gertz, G. Saake: *Transitional Monitoring of Dynamic Integrity Constraints*. Bull. IEEE Technical Committee on Data Engineering 17:2, June 1994.
16. B. Meyer: *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.
17. B. Meyer: *Applying "Design by Contract"*. IEEE Computer 25:10, 1992.
18. J.E.B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
19. T. Sheard, D. Stemple: *Automatic Verification of Database Transaction Safety.* ACM Trans. on Database Systems 14:3, 1989.

Appendix: Consistency Constraint Checking vs. Transaction Proofs

Practical schemas can define a huge number of constraints, which all will be checked at certain points in time. Poor performance can be the result, although some constraints can be known to hold always. Therefore, it is not necessary to check them at runtime. On the other hand, deleting CCs from class and method definitions is not desirable, since these CCs still are beneficial for documentation purposes. We thus allow to switch single constraints on or off. Switching a constraint off means that it will no longer be checked at runtime, until it is switched on again. Note that even parts of invariants, pre- and postconditions can be deactivated, as these parts can be referred to via their name.

Syntactically, CCs can be switched on and off interactively with the commands:

```
assume name in class_name      // switch off consistency constraint checking for name
check name in class_name       // switch on consistency constraint checking for name
```

These two commands would be implemented through the **activate** and **deactivate** operations provided by SAMOS.

Note that in this way our PbC-based approach can be combined with approaches that perform *transaction verification* at compile time [e.g., 18, 11]. Application programs and methods can be successively verified against the specified CCs, which can then be deactivated. Concretely, the following implementations have to be checked before the constraints can be switched off:

- after the precondition of a method (or application program) has been proven to be guaranteed by each client (sender of the corresponding message), the precondition can be deactivated,
- after the method has been proven to obey its postcondition, the postcondition of this method can be switched off,
- after each method implementation has been verified with respect to the invariant of the corresponding class, the invariant can be deactivated.