Secondary Publication

Henrich, Andreas; Däberitz, Dirk

Using a query language to state consistency constraints for repositories

Date of secondary publication: 14.02.2025

Accepted Manuscript (Postprint), Conferenceobject

Persistent identifier: urn:nbn:de:bvb:473-irb-1063793

Primary publication

Henrich, Andreas; Däberitz, Dirk (1996): Using a query language to state consistency constraints for repositories, in: Roland R. Wagner und Helmut Thoma (Ed.), Database and expert systems applications : 7th international conference, DEXA '96, Zurich, Switzerland, September 9 - 13, 1996 ; proceedings, Berlin u. a.: Springer, pp. 59–68, doi: 10.1007/BFb0034670.

Legal Notice

This work is protected by copyright and/or the indication of a licence. You are free to use this work in any way permitted by the copyright and/or the licence that applies to your usage. For other uses, you must obtain permission from the rights-holders.

This document is made available with all rights reserved.

Using a Query Language to State Consistency Constraints for Repositories

Andreas Henrich and Dirk Däberitz

Praktische Informatik, Fachbereich Elektrotechnik und Informatik,
Universität Siegen, D-57068 Siegen, Germany,
e-mail: {henrich|daeberitz}@informatik.uni-siegen.de

Abstract. Modern system development environments usually deploy the object management facilities of a repository to store the documents created and maintained during system development. In this paper we present a pragmatic approach to express various levels of consistency for the data maintained in such a repository. The levels of consistency are achieved by means of a query language. Consistency constraints are stated as queries searching for inconsistent items in the repository. The queries are handled by a generic analyzer, which checks the defined constraints whenever appropriate. Furthermore, we present a general classification schema for consistency constraints and describe how different classes of constraints are handled by our approach.

1 Introduction

Repository-based applications are in widespread use in the domain of system development environments. PCTE is the ISO and ECMA standard for a public tool interface for an open repository [15]. PCTE contains as one of its major components a structurally object-oriented OMS. The standard covers aspects like the navigational access to the data, access rights, replication, distribution or schema management. The main objective of the OMS is data integration.

On the other hand, quality and quality assurance are of paramount importance in system development and if project data is maintained in an open repository, it is natural to define constraints to improve the quality of the data. Useful constraints range from naming conventions to balancing criteria between different diagrams or graphical constraints corresponding to the layout of a diagram.

Although PCTE provides a semantically rich data model, the data model is not expressive enough to represent all conceivable consistency constraints (CCs) – and the same is true for all other OMSs we are aware of. Most CCs require advanced computations or access to other entities of the OMS, and especially in system development environments intermediate states of a document may occur violating certain CCs which must be fulfilled by the final document. Some CCs may e.g. be inappropriate or unnecessary during the creation or early states of a document - we call these increasing quality requirements *"levels of consistency"*.

The first aspect of computation and access to other entities is addressed to a certain extent by persistent programming languages or application specific tool

definition languages [7]. These solutions introduce a very complex data model comparable to programming languages. Unfortunately these complex features complicate the simple modeling process and the second problem – different levels of consistency – is not considered by the complex specification languages yet.

In [14] a software architecture is presented allowing application programs to access documents at various levels of consistency, but once the programs are realized and executed they are bound to a certain level of consistency.

Other research projects investigate declarative languages for the specification of CCs. A lot of papers have been presented to this topic in recent years (see [1, 3, 4, 9, 11]). However, these solutions are related to explicit constraint definition languages. They provide a look to the problem from the database kernel, whereas our approach takes a look from the area of database application.

We present an application oriented approach to overcome the mentioned problems by using a query language to define CCs. Our idea is to specify a CC by a query which extracts those items which do not satisfy the CC. The items are processed by an application program – we call it "analyzer". To realize various levels of consistency, we specify various queries. With other words we define a consistency level as a set of CCs which should be satisfied by the OMS.

2 Example Environment

PCTE: PCTE, Portable Common Tool Environment, is the ISO and ECMA standard for a public tool interface (PTI) for an open repository [15]. The PCTE data model can be seen as an extension of the binary Entity-Relationship Model. The objectbase contains objects and relationships. Relationships are normally bi-directional. Each relationship is realized by a pair of directed links, which are reverse links of each other, i.e. point into opposite directions.

The type of an object is given by its name, a set of applied attribute types and a set of allowed outgoing link types. New object types are defined by inheritance.

A link type is given by a name, an ordered set of attribute types called key attributes, a set of (non-key) attribute types, a set of allowed destination object types and a category. PCTE offers five link categories: *composition* (defining the destination object as a component of the origin object), *existence* (keeping the destination object in existence), *reference* (assuring referential integrity and representing a property of the origin object), *implicit* (assuring referential integrity) and *designation* (without referential integrity).

Throughout this paper we will use the simplified schema for OOA-documents given in figure 1. The object types are given in rectangles. The attribute types applied to each object type are given in the ovals at the upper left corner of the rectangles representing the object types.

The link types are indicated by arrows. A double arrowhead at the end of a link indicates that the link has cardinality *many*. Links with cardinality many must have a key attribute. In the example the numeric attribute *number* is used for this purpose. For example the link type *contains* has such a key attribute and is hence described as *"number.contains"*. Therefore an instance of this link

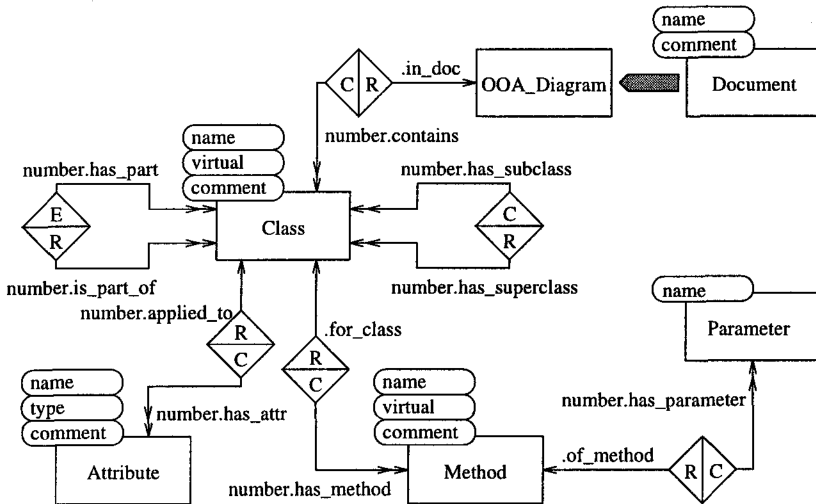

Fig. 1. Example schema

type can be addressed by its link name which consists of the concrete value for the key attribute and the type name separated by a dot – e.g. *"3.contains"*. A *'C'*, *'E'* or *'R'* in the triangles at the center of the line representing a pair of links, indicates that the link has category *composition*, *existence* or *reference*.

Finally the schema contains a subtype relationship between the object types *Document* and *OOA_Diagram* which is indicated by a broad shaded arrow.

P-OQL: P-OQL is an OQL-oriented query language for (H-)PCTE [12]. The main differences to standard OQL [2] are due to the adaptation to the data model of PCTE. Hence, especially the treatment of links is specific to P-OQL.

A query in P-OQL is either a select-statement, or the application of an operator like *count*. As with SQL a select-statement consists of three main parts:

`select` *target-clause* `from` *base-sets-clause* `where` *qualification-clause*

Assume that we search for the name and the methods of all classes in the OOA-diagram named *"Billing"* for which the instances can be used as parts:

```
select C:name, C:_.has_method/->name
   from D in OOA_Diagram,
        C in (D:_.contains/[_.has_subclass]*/->.)
   where D:name = "Billing" and
         0 < count C:_.is_part_of->.
```

In the **base-sets-clause** of this query two base sets are defined: Base set `D` addressing all OOA-diagrams and base set `C` addressing all objects which can be reached from the actual object of base set `D` via a path matching the regular

path expression _.contains/[_.has_subclass]*. In the definition of base set C 'D:' means that the actual element of base set D is used as starting point of the definition. '_.contains' means that exactly one link of type *contains* must be traversed. Here '_' is used as a wild card for numerical key attributes denoting that arbitrary key values are allowed. '[_.has_subclass]*' means that zero or more links matching the link definition _.has_subclass have to be traversed. Alternatively P-OQL knows [*path_definition*]+ to indicate that a path matching *path_definition* has to be traversed at least once and [*path_definition*] to indicate that a path matching *path_definition* is optional. '/->' is used to address the destination object of the path. In addition '->' can be used to address the last link of the path. The '.' at the end of the regular path expression means that the object (or link) under concern is addressed. It is also possible, to address an attribute – which is e.g. the case in the target-clause of the example query above – or to address a tuple of values (see [12] for more details).

In addition to the link definitions used in the above example, which have been based on a given link type name and a definition of the allowed values for the key attributes using wild cards or intervals, P-OQL allows the specification of a set of link categories instead, with the meaning that all links having one of the given categories fulfill this link definition. E.g. the expression [{c, e}]+/->. addresses all objects which can be reached via a path consisting only of links with category *composition* or *existence*. Furthermore, not only the category of the link itself, but also the category of its reverse link can be specified using '@' as a prefix for the category definition.

Summarizing, a base set can be defined in P-OQL in five different ways: (1) giving an object type name or an object type name suffixed by a '^' meaning that objects of all subtypes are addressed as well; (2) using a link type name to address all links of a given type; (3) defining a set of objects or links using a regular path expression as with base set C in our example; (4) defining a set of objects or links via a sub-select; or (5) passing a set of objects or links via the API when submitting a query.

In the **qualification-clause** of our example query two conditions are given; D:name = "Billing" and 0 < count C:_.is_part_of->.. In the second condition a regular path expression is used to address the set of *is_part_of* links originating from the class under concern. This set must not be empty.

The **target-clause** of the example query defines, that a bag of pairs is requested. Each pair consists of (1) the name of the class and (2) a bag with the names of the methods of the class.

3 A Classification Schema for Consistency Constraints

To state precisely which types of CCs are covered by our approach, we present a classification schema for CCs which is summarized in table 1.

Scope: The scope of a CC describes the number of items, which have to be examined to check the CC. An *atomic* scope CC means that only a single item has to be inspected. A simple example is the CC *"A class in an OOA-diagram*

dimension	possible values
scope	atomic | structure | extension
execution mode	periodical | at request (implicit | explicit)
realization mode	complete | incremental
subject	value | structure | method
location	external | internal (inherent | implicit | explicit)

Table 1. A classification schema for consistency constraints

must have a nonempty name!". A *structure* scope CC has to examine more than one item within a defined structure to satisfy the CC. Imagine a CC like *"All classes in an OOA-diagram must have a different name!"*. For this CC all classes within a certain OOA-diagram have to be checked. The scope of the CC is the OOA-diagram. An *extension* scope CC depends on all items of a given type, i.e. the extension of the type is the scope of the CC. A restrictive CASE environment could e.g. enforce the following CC: *"All OOA-classes in the repository must have different names!"*.

Execution Mode: The execution mode describes at what points in time the CC is checked. A first execution mode is *periodical* meaning, that the CC is checked at fixed points in time. Another execution mode is *at request*. The execution point is not foreseeable and usually triggered by a tool user or an asynchronous external event. There are two possibilities to execute a CC at request. Either the CC is executed *explicitly*, that means the CC is executable separately and needs explicit trigger mechanisms, or the CC is executed *implicitly*, that means it is checked whenever modifications of data in the repository occur.

Another opportunity would be to integrate CC checking with a sophisticated transaction mechanism enabling long end nested transactions. In this case the CCs would be checked at the end of selected transactions (EOT). However, we did not consider this approach in more depth because the coupling between EOT and CC checking restricts the flexibility of the tool user. E.g. a tool user may be interested in the objects violating certain CCs in an intermediate state to decide whether to fix the existing violations immediately or to delay their consideration.

Realization Mode: In [8] the *realization mode* of a CC is introduced. If the realization mode is *incremental*, only those items of the repository are checked, which have been changed. This may include not only the modified item itself but also a certain defined context, e.g. a name space. The other realization mode is *complete*. A complete realization scans a large part of the database to satisfy the CC. This part is fairly unrelated to the items which have actually changed and may even comprise the whole database.

Subject: Another criterion in our classification schema is the *subject* of a CC. With a *value based* CC, the subject are one or more attribute values. That means the CC is expressed in terms of attribute values and comparisons of these values. With a *structure based* CC, the subject is the structure of the object

graph. Based on the example schema given in figure 1 such a CC could e.g. be *"There must not be a cycle in the class hierarchy defined by inheritance!"*.

Another possible subject would be the *behavior* of an object. Behavioral CCs allow to check the correct semantics of methods applied to an object type [10]. Because the data model of the example OMS PCTE does not allow the specification of behavior we do not consider behavior based CCs in this paper.

Location: Another dimension of CCs is discussed in [10]. *Internal* CCs are those that are known to and can be handled by the DBMS, while *external* CCs have to be expressed, checked and enforced within application programs.

Internal CCs can be subdivided into the following types: (1) *Inherent* CCs are fixed for the data model of the DBMS. (2) *Implicit* CCs are expressed by means of the data modeling facilities of the DBMS. (3) *Explicit* CCs cannot be expressed by means of the data modeling facilities of the DBMS. They are maintained by a separate CC definition language.

Another classification dimension for CCs could be based on whether only single database states or even database state transitions can be constrained [6]. However in this paper we deal only with single database states.

4 Using Queries to state consistency constraints

This chapter presents a mapping from our classification schema to the features of the PCTE-OMS and the query language P-OQL.

Location: The *location* criterion is somewhat different from the other dimensions of our classification schema, because it classifies the CCs with respect to the feature of the database used to implement them. With respect to this criterion, our approach to use of a query language to state CCs is located somewhere between *internal explicit* and *external* CCs. On the one hand, the use of a query language allows to state the CCs in a compact declarative way. On the other hand, the enforcement of these CCs is not automatically done by the DBMS. Instead, the application can either initiate a check explicitly (on user demand or periodically) or use the notification mechanism of the OMS to enforce a check when attribute values or links are changed for an object (this will be discussed in more detail for the *execution mode*).

Scope: For an *atomic scope* CC, only a single object has to be inspected. A query representing such a CC must return the objects violating the CC.

Example 1. A typical *atomic scope* CC is *"There are at most 7 parameters per method!"*. The corresponding query looks as follows (In contrast to the example in section 2 the base set in this query has no name because there are no ambiguities.):

```
select . from Method where 7 < count _.has_parameter->.
```

A regular path expression is used to calculate the set with all originating links of type *has_parameter* for the *Method* object under concern. The cardinality of this

set is calculated using the *count*-operator. The dot in the target-clause means, that an object reference should be returned for each object violating the CC. □

In contrast to an *atomic scope* CC, a *structure scope* CC, cannot be checked for an object itself, but by comparing the object with other objects in the structure. Therefore, tuples of objects violating the CC are returned in this case.

Example 2. To check the CC *"All classes in an OOA-diagram must have different names!"* we have to address for each class the diagrams containing the class and all other classes contained in these diagrams. One way to proceed would be to use the regular path expression `[_.has_superclass]*/.in_doc/->.` to address the OOA-diagram(s) containing the class and to use the regular path expression `_.contains/[_.has_subclass]*/->.` to address the classes contained in each achieved OOA-diagram. Whereas this use of regular path expressions would yield expressions specific to the example CC, we can also use a more generic query formulation based on category definitions:

```
select A:., B:.
   from A in Class, D in (A:[{@c}]+/->.), B in (D:[{c}]+/->.)
   where D:. is of type OOA_Diagram and
         B:. is of type Class and
         A:name = B:name and A:. < B:.
```

The query works as follows: For each class (base set `A`) the objects containing this class are addressed in base set `D` using the regular path expression `A:[{@c}]+/->.`. In the qualification-clause this base set is restricted to OOA-diagrams. For each of these OOA-diagrams objects which can be reached via *composition* links are addressed in base set `B` using the regular path expression `D:[{c}]+/->.`. In the qualification-clause this base set is restricted to classes.

Hence, each class a is combined with each class b which is contained in an OOA-diagram together with a. For those pairs we check if they have equal names. Furthermore, we check that a and b do not refer to the same object. To this end, we exploit the fact, that P-OQL has a '<'-operator for objects. □

Finally we have to map *extension scope* CCs to the query language. Here the objects, which have to be compared, do no longer depend on the structure.

Example 3. We remember the CC *"All classes in the repository must have a different name!"*. This can be stated as follows:

```
select A:., B:.
   from A in Class, B in Class
   where A:name = B:name and A:. < B:.
```

In this query each class is compared with each other class in the objectbase. □

Realization Mode: In the above section, we have assumed the realization mode *complete*. However, this may lead to serious problems. On the one hand,

performance problems may arise due to the size of the objectbase and on the other hand this does not take into account, that some OOA-diagrams may be in an intermediate state where certain CCs should not be enforced. Therefore it will be much more appropriate to use an *incremental* realization in most cases. To this end, passed base sets can be used in different ways. (1) Whenever an object is changed a one-element base set with this object is created and passed to the query checking the CC. (2) Changed objects are collected in a sequence and when finishing a certain amount of changes, the relevant CCs are checked by a query based on this sequence. (3) When the editing of a certain document is done, the objects of this document are checked against the relevant CCs.

Example 4. Assume the simple CC *"There are at most 7 parameters per method!"* from example 1. We can easily realize an incremental version of this query exchanging the base set defined by the object type *Method* with a passed base set `OBJECTS[0]` addressing a sequence of objects passed to the query processor together with the query itself via the API:

```
select . from OBJECTS[0] where 7 < count _.has_parameter->.
```

Of course this query assumes that only objects of type *Method* are passed. □

If all changed objects are contained in the passed base set, a condition checking the object type must be added to the qualification-clause:

```
select .
   from OBJECTS[0]
   where . is of type Method and 7 < count _.has_parameter->.
```

It is also possible, to state a query which checks a CC for the objects in one or more documents, which have changed. To this end, all components of the expected type in the document have to be addressed similar to example 2.

Execution Mode: Another classification dimension was the execution mode of a CC. A *periodical execution* seems to be inappropriate in the area of CASE, because some parts of the repository will not change over a long period of time while others are changing rapidly. Therefore it is more suitable in this application area to execute CCs at request, e.g. if data is modified.

If the execution mode is *explicitly at request*, the application has to offer a facility to start the checking of some CCs to the user. In section 5 we will sketch this facility in the generic analyzer.

If the execution mode is *implicitly at request*, the application has to check the CCs after each change of the repository. Since repositories like the OMS of PCTE offer powerful notification mechanisms we can realize this in a generic way executing changes in transactions and connecting the CC checker with the notification mechanism (see [13] for more details).

Subject: In the previous sections we have already presented simple examples for *value based* and *structure based* CCs. A more sophisticated example for a *structure based* CC would be a cycle test.

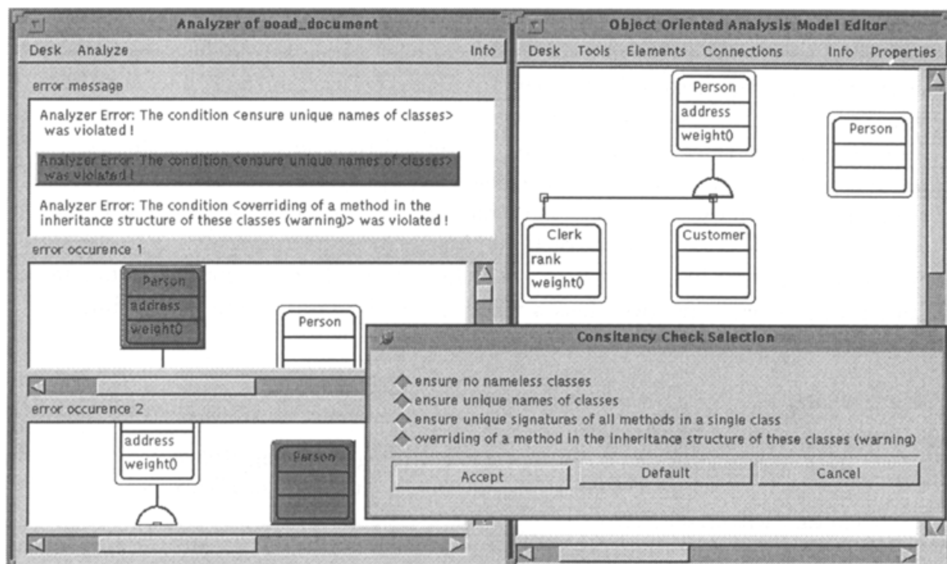

Fig. 2. An OOA/OOD-analyzer (screen dump)

Example 5. The CC *"There must not be a cycle in the class hierarchy defined by inheritance!"* can be stated as a query in the following way:

```
select . from Class where . in [_.has_subclass]+/->.
```

In the qualification-clause of this query the regular path expression `[_.has_subclass]+/->.` is used to calculate a set containing all direct and indirect subclasses of the class c under concern. If c itself is in this set, the class hierarchy includes a cycle. □

5 Implementation Issues

As a result of the research work presented in this paper we have realized a generic analyzer, based on the generic software architecture presented in [5]. The generic analyzer is controlled by resources. Each resource contains a set with queries for a certain object type. Each query in a resource represents a certain CC. The analyzer is invoked on a single object in the OMS, examines its object type and loads the corresponding resource. The user of the analyzer may disable some of these queries to set various levels of consistency for the object type under concern. The maintenance of the resources is quite simple and can be done using a text editor. The addition of new queries, respectively new CCs, is as easy as an adaptation of existing queries to new schema definitions in the repository. Figure 2 shows a screen dump of our generic analyzer.

The right hand side of the screen dump shows an OOA/OOD-editor, the left hand side shows the generic analyzer, which is controlled using a resource containing four queries to be checked for OOA/OOD-diagrams. The small dialog window allows the user to enable the four queries separately. To provide a convenient user interface the queries are presented by a short note to the user. The analyzer itself consists of three sections. The upper section shows a list of violations of CCs. An error message is given here for each violation. The selection of a message leads to an update of the error occurrence views, which are situated below the message list. The occurrence views show the erroneous objects – in our screen dump these are two classes which have the same name in the diagram.

References

1. C. Bauzer-Medeiros and P. Pfeffer. Object Integrity Using Rules. In *Proc. European Conf. on Object-Oriented Programming*, volume 512 of *LNCS*, pages 219–230, Geneva, Switzerland, 1991.
2. R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, Calif., USA, 1993.
3. S. Ceri, P. Fraternali, and S. Paraboschi. Constraint Management in Chimera. *Data Engineering Bulletin*, 17(2):4–8, 1994.
4. S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proc. 16th Intl. Conf. on VLDB*, pages 566–577, Brisbane, Australia, 1990.
5. D. Däberitz and U. Kelter. Rapid prototyping of graphical editors in an open SDE. In *Proc. 7th Conf. on Software Engineering Environments*, pages 61–72, Noordwijkerhout, 1995.
6. O. Deux. The O2 System. *Comm. of the ACM*, 34(10):34–48, October 1991.
7. W. Emmrich. *Tool Construction for Process-Centered Software Development Environments based on Object Databases.* University of Paderborn, Germany, 1995.
8. G. Engels and W. Schäfer. *Program Development Environments - Concepts and Realization.* Teubner, Stuttgart, 1989. (in German).
9. S. Gatziu and K.R. Dittrich. Events in an Active Object-Oriented Database System. In *Proc. 1st Intl. Workshop on Rules in Database Systems*, Workshops in Computing, pages 23–39, Edinburgh, Scotland, 1994.
10. A. Geppert and K.R. Dittrich. Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming By Contract. In *Proc. GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft"*, Springer, Informatik aktuell, pages 322–337, Dresden, Germany, 1995.
11. P.W.P.J. Grefen, R.A. de By, and P.M.G. Apers. Integrity Control in Advanced Database Systems. *Data Engineering Bulletin*, 17(2):9–13, 1994.
12. A. Henrich. P-OQL: an OQL-Oriented Query Language for PCTE. In *Proc. 7th Conf. on Software Engineering Environments*, pages 48–60, Noordwijkerhout, 1995.
13. Andreas Henrich and Dirk Däberitz. Using a query language to state consistency constraints for repository-based system development environments. Interner Bericht 96/3, Universität Siegen, 1996.
14. R. Marti. *An Approach Towards the Design of Software Development Systems.* Verlag der Fachvereine, Zürich, Schweiz, 1994. (in German).
15. Portable Common Tool Environment - Abstract Specification / C Bindings. Standards ECMA-149/-158, 3rd edition, and ISO IS 13719-1/-2, 1995.