# Organizing Software Repositories
# Modeling Requirements and Implementation Experiences

Thomas Rose
Computer Science Dept.
University of Toronto
Toronto, Ont.
Canada, M5S 1A4

Matthias Jarke
Informatik V
RWTH-Aachen
D-W-5100Aachen
Germany

John Mylopoulos
Computer Science Dept.
University of Toronto
Toronto, Ont.
Canada, M5S 1A4

A version of this paper appears in D.B. Simmons (ed.): *Proc. of the 16$^{th}$ Intl. Computer Software & Applications Conf.* (CompSAC '92), Chicago, IL, September 23-25, 1992.

***Abstract***

*Software repositories should not only provide the service of managing evolving objects. Repository technology can also be used to maintain the consistency among software objects and software-related descriptions in integrated environments. This requires addressing the representational adequacy and semantics of present object management systems. Based on experiences gained in a series of CASE integration projects, this paper points out abstractional, assertional and dynamic clustering requirements of a data modeling language for repository managers. A simple example illustrates how the deductive object management system ConceptBase embodying Telos as its data model meets these requirements.*

***Keywords:*** *software engineering databases, software repositories, object management systems, data models, conceptual modeling, knowledge representation*

## 1 Introduction

Repository technology has its technical roots in library support inside software development environments. In the past, libraries have been deployed for documenting and archiving services. As such, repositories have for the most part resembled file cabinets gathering and archiving documents produced throughout the lifetime of a software product. Each file handles a project, i.e. it manages an evolving program.

More recently, repository technology is being used to integrate environments layered around object management systems. The focus of attention moves from merely *capturing* data to the *sharing* and *integration* of data. Repositories become first-class tools of an environment. The role of repositories matures to maintaining data consistency and disseminating objects and dependencies inside an environment, in other words, the repository becomes a full-service broker of an environment's clearinghouse. This requires natural descriptions and annotations of objects and relationships. Thus, the question of feasible data models emerges. So, what then are useful data modeling facilities ?

Instead of considering the source documents of software objects directly, we focus on maintaining descriptions and annotations of these objects and their relationships. Actual textual or graphical documents — e.g. program sources, designs, or specifications — are referred to by references to external tools which manage evolving objects in a network. This has less to do with separating out functionality, but has more to do with pointing out distinct concerns. To allow the integration of heterogeneous tools and data sources, a repository should be perceived as a *two-level information system* with a *conceptual knowledge* level and a *physical representation* level, the documents.

This paper points to the representational requirements for capturing and processing the conceptual knowledge about software descriptions. These requirements were derived from projects that required the conceptual integration of heterogeneous environments in the domains of computer-assisted software engineering [15, 17, 9], version and configuration management [29], decision-based change propagation [28], computer-integrated quality management [16], hypertext co-authoring [11, 31], and teamwork support [12]. Although the sorts of objects and dependencies are different, there are three common lines of requirements: *abstractional*, *assertional* and *dynamic clustering facilities*.

The idea of different kinds of abstraction for capturing the application domain stems from conceptual modeling [6]. Yet, conceptual modeling itself is insufficient, since it is limited to structural properties for the most part. Inter- and intra-object dependencies require assertions to constrain software objects and rules to express inferred relationships among software objects and dependencies. Moreover, dynamic clustering emerges as a vital feature too, because objects frequently change classifications and move from one "pool" to another. During the course of our projects, it has turned out that each facility is necessary and that a coherent framework is useful.

For the most part, emerging commercial repository proposals rely on adaptations of the Entity-Relationship model [8]. For instance, AD/Cycle's repository manager adheres to the concepts of entity and relationship, and provides abstractions to deal with generic aggregations [22, 30]. Yet, they provide little support for either assertions or clustering. Active databases incorporate triggered procedures [7]. This is certainly a step forward but has the problems of "grasping" the semantics and checking the consistency of repository procedures. Considering the central role of the repository in an enterprise, these problems, in the long run, are not tolerable.

One way to reduce these representational deficits have been recent attempts to integrate concepts from object-oriented and deductive databases. Our approach is along these lines applying a knowledge representation language called *Telos* [24] and its implementation in a deductive object base, *ConceptBase* [14].

---

This work is part of a project titled "Development of Intelligent Information Systems" and is funded by the Federal Networks of Excellence programme through the Institute of Robotics and Intelligent Systems (IRIS). Parts of this work were also supported by the Deutsche Forschungsgemeinschaft under Grant Ja-445/1-2 and the Commission of the European Communities within ESPRIT project 892 (DAIDA).

Section 2 identifies the abstractions and dependencies that need support, using a simple example for illustration. The example concerns program sources, code files, and compilation. Section 3 introduces the modeling facilities of *Telos* and the support facilities of *ConceptBase* through the example. To conclude, the final sections relate our approach to other work and set forth its potentiality.

## 2 Requirements of Repository Data Models

A library serves as central documentation facility. It allows one to maintain and exchange software objects among programmers and projects. For instance, suppose we have a library to keep track of program and code files.

> ***Example*** *(program source, code and compilation)*
>
> *The domain consists of a bunch of files containing program sources and executable code. Program files are assumed to be the units of development which undergo frequent modifications. A compilation establishes a relationship between a program and a code file. Furthermore, compilations have to adhere to dependencies which are due to the structure of a software system as represented by "uses"-relationships. In general, compilation of one source results in necessary recompilations of other sources because of "uses"-relationships. Since compilations are expensive, changing one source should result in as few as necessary recompilations to maintain the system's consistency.*

To start with, one has to capture the entities and the relationships that a software development environment has to maintain. The entities of our example are program and code files.

***Classification*** *(classes and metaclasses)*

In general, program sources exist in versions. Hence, organizational needs require one to classify. Take program sources of an editor for instance. They all belong to an editor class which is an abstraction describing properties common to any version. Classes serve two purposes. On the one hand, a class is a conceptual abstraction describing properties of editors that any instance of an editor must have. On the other hand, it manages each version of an editor as a class instance.

In general, sources and codes have different properties, e.g. sources can be edited while codes can not. This kind of abstraction should take place at a metalevel which is a natural place to make statements about classes. Thus, metaclasses are useful to represent properties of classes of entities and relationships. They also allow one to exchange properties among classes. Take a *date-of* attribute for instance. Does it refer to the date of creation or the date of last modification? If one tool conceives it one way and another tool the other way, the worst case emerges — an object's property changes semantics when crossing applications.

***Specialization*** *(inheritance)*

Although sources and codes have little in common, they are both documents sharing features like *filename*, *date-of-creation* or *owner*. Such features are not specific to sources and codes but feature documents in general. Documents that specialize to source and code files inherit generic properties from documents. Source and code representations are refined by adding additional features which are unique to each. Typically, source files have a reported version history while code files do not. Thus, abstractions for specialization and inheritance are necessary.

***Flexible Aggregation***
*(classification, specialization & attribution of attributes)*

Attributes represent relationships among objects. Some of them merely represent values, like *date-of-creation*. Others embody much more information. At first glance, a compilation merely constitutes a relationship between a source and code file showing how to get the code for a given source file. Hence, representation in terms of an attribute attached to program sources sounds feasible. But, a compilation is done at a specific date with a specific compiler version, it succeeds or fails, generates a report, etc. Moreover, representing specializations and classifications of compilations sounds reasonable too, since a compilation might be specialized to produce code for customer delivery or code including debugger options. Hence, a compilation resembles a full-fledge entity in its own right. Consequently, attributes capturing relationships should become first-class objects, which are a matter of attribution, classification and specialization.

So far, the representation considers structural properties. It does not account for deeper properties, e.g., the consistency of objects and relationships which is of particular concern in configuration management. Then, what facilities are needed to specify consistent sets of program and code files ?

***Static Dependencies*** *(integrity constraints)*

Roughly speaking, a set of modules `M` is said to be consistent, if for each module $M_i$ of `M` any required service is provided by a module $M_j$ ($i \neq j$) in `M`. For the most part dependencies are expressed in this way. Often, they are just "presented" in manuals and maintained by tools. Usually, underlying axioms about dependencies are not pointed out. Meyer presents a set of inter- and intra-program dependencies whose

consistency is expressed in terms of set-oriented axioms [23]. Although they embody a fundamental property, they cannot be captured by structural abstractions. Such axioms are better represented in logic for reasons of precision and comprehension. Integrity constraints naturally provide a sound assertional facility and allow us to express statements about what dependencies are of importance.

***Dynamic Dependencies*** *(deductive rules)*

Other dependencies concern dynamic relationships. A recompilation of module `M_i` depends on a modification of module `M_j`, if `M_i` uses services provided by `M_j`. The code of `M_j` becomes obsolete and so `M_i` has to be recompiled too. Thus, `M_i` depends on `M_j`. Since `M_j` might depend on another module `M_k`, `M_i` becomes dependent on `M_k`. Eventually, a modification causes the recompilation of a set of modules. Actually, dependencies of this type are static, since they depend on "uses"-relationships between modules. Deploying smart recompilation [33], necessary and sufficient recompilations are determined by "uses"-relationships *and* the modification done to a module. Thus, different modifications to a module result in recompiling different modules to maintain consistency.

Hence, dependencies result in recursive structures which vary dynamically with respect to object properties, dependencies and modifications. Again, logic is the natural way to achieve a precise and comprehensive representation. Deductive rules are useful to capture the underlying axioms of dependencies and to maintain them dynamically.

So far, we have discussed abstractional and assertional requirements of a repository data model. However, the deployed concepts lack flexible classification facilities; they do not account for dynamic restructuring and flexible clustering of data.

***Dynamic Clustering*** *(parameterized derived classes)*

Usually, programs are assigned to programmer teams and labeled with respect to development stages, such as *changing*, *testing* or *released*. For reasons of maintenance, any document which has been released and distributed to a customer might be archived. Hence, an archive is defined as a cluster which crosses class boundaries. Membership to this cluster is determined by an attribute value, e.g. each object has property *released*. Members of an archive change dynamically, e.g. with respect to movements among development stages. Thus, derived classes become useful to gather instances of other classes. They are generalizations or specializations of existing classes. Membership is not by storage but by inference. Membership is specified by rules which make use of structural properties (e.g., it is a program), as well as, assertions (e.g., the value of attribute *development-stage* is *released*).

Derived classes support flexible classification. Still, a focusing facility is missing. Imagine a manager who is monitoring a project's progress. At a glance, he is anxious to see which programs have been released, in other words, what parts of the project have been finished. He might also want to screen what programs were released last month and the month before, or he might be curious about single team's performances. Thus, parameterization of derived classes is natural. The membership specification contains a parameter, like authors of programs and date of release.

To conclude, when developing repository data models one should consider three lines of requirements: *abstractional*, *assertional* and *dynamic clustering* facilities.

## 3 Repository Modeling with Telos in ConceptBase

*Telos* [24] is a hybrid conceptual modeling language having its roots in semantic data modeling and knowledge representation. By amalgamating abstractional facilities of object-oriented languages and assertional facilities of deductive databases Telos provides a modeling framework that comes close to the requirements of section 2.

### 3.1 Abstractional Facilities

Telos supports three lines of abstraction: *classification*, *generalization* and *attribution*. Fig. 1 shows a Telos repository model for documents using the notions of aggregation, classification and specialization. A document description features an `owner`, a `date-of-creation` and an `allocation`. The actual document is handled by a document management system; `allocation` serves as reference for external identification. A `Program` specializes a `Document` by additional properties and relationships to other documents, like `uses`, `version-of` and a reference to a corresponding code file (`compiled-to`). `Code` specializes `Document` by attribute `depends-on` which is intended to capture the dependencies among produced code files.

```
Individual Document in Class with
  attribute
    owner: Agent;
    date-of-creation: Date;
    allocation: File
end Document

Individual Program isA Document with
  attribute
    version-of: Program;
    successor: Program;
    last-modification: Date;
    in-state: Stage;
    uses: Program;
    compiled-to: Code
end Program

Individual Code isA Document with
  attribute
    depends-on: Code
end Code
```

**Fig. 1:** Structural representation

While aggregation and classification set up the structural relationships among programs and codes, attribute refinements capture further properties of relationships. Since Telos treats attributes as first-class objects, it supports attribution, classification and specialization of attributes themselves. Fig. 2 shows a fine-grained description of the relationships `version-of` and `compiled-to`, which are both captured by attributes of `Program`. Syntactically, an attribute is identified by a `!`-notation. `Program!version-of` denotes the full-fledge *object* which stands for attribute `version-of` of class `Program`; `Program.version-of` denotes the *value* of attribute `version-of`.

A program version is produced by an agent at a specific time. One may want to document the purpose of this change by a comment. Similar to versioning, a compilation is done at a specific time and produces a compilation report. As such, an attribute refinement yields a smarter representation of those relationship which the attribute is supposed to capture. Information is attached to the attribute capturing the relationship rather than encoded in objects involved in the relationship.

```
Attribute Program!version-of with
  attribute
    at: Date;
    by: Agent;
    comment: String
  constraint
    proper-access-right:
      $ forall agent/Agent prog/Program
            prog.owner = agent ==>
                prog!version-of!by = agent $
end Program!version-of

Attribute Program!compiled-to with
  attribute
    at: Date;
    error-flag: CompilerFlag;
    report: Document
end Program!compiled-to
```

**Fig. 2:** Capturing properties of relationships

In general, treating attributes as full-fledged objects allows flexible adaptations without affecting the genuine structure. While the basic structure remains stable, attributes, integrity constraints and inferred attributes can be adjusted in order to capture change inside the environment.

### 3.2 Assertional Facilities

Relationships can be constrained by integrity constraints. Constraint `proper-access-right` limits the versioning of a program to the program's owner (fig. 2). Proper use of `version-of` requires that a new version be produced by the owner of the old version.

Program versions are handled in a double-linked chain. While links for backward chaining are stored, forward links are inferred. This is an arbitrary decision based on the assumption that there are no preferences. Hence, `version-of` and `successor` are redundant, with `successor` specified in terms of a deductive rule (fig. 3).

```
Individual Program isA Document with
  rule
    derive-last-modification:
      $ exists d/Date
          this!version-of.at = d
              ==> this.last-modification = d $
    derive-successor:
      $ forall prog/Program
          prog.version-of = this ==>
              this.successor = prog $
end Program
```

**Fig. 3:** Deriving and maintaining redundant data

The representation of `Program` depicted in fig. 3 refines the one given in fig. 2 by adding knowledge. `Derive-successor` computes the successor of a program version, if any is available . Similar to version links, date of `last-modification` is derived from the date `at` of the attribute representing the change. Hence, deductive rules are feasible to maintain

redundant attributes consistent. Consequently, it is no more the task of a tool to update a bunch of attributes consistently, possibly spread across a schema.

In logical assertions of rules and constraints the variable `this` refers to each instance of a class as a shorthand, i.e. here `this` acts as a variable `program` which is quantified by `forall program/Program`.

### 3.3 Dynamic Clustering Facilities

An extension to Telos, ConceptBase supports dynamic clustering by so-called *query classes* [18]. Query classes specify and maintain "virtual" classes. Query classes are actually classes from an abstractional point of view but with a different classification behavior. Membership is specified by deductive rules and constraints. For instance, released programs constitute a subset of the set of programs as specified by `ReleasedProgram` (fig. 4). The query rule `is-released` reads, each program is a member of `ReleasedProgram` whose attribute `in-state` has value `Released`. Actual members of `Released-Program` vary in general. Any time a program moves to state `Released`, it becomes a member of `ReleasedProgram`. The update of query classes is done automatically, that means, again no tool is necessary for maintaining memberships to clusters.

```
QueryClass ReleasedProgram isA Program with
  parameter
    producer: Agent
  query_rule
    is-released:
      $ this.in-state = Released and
          this.owner = producer $
end ReleasedProgram
```

**Fig. 4:** Dynamic clustering

The flexibility in specifying clusters is enhanced by a parameter facility, such as parameter `producer` in `ReleasedProgram` (fig. 4). A parameter allows one to adjust clusters by passing particular objects which are of concern. Without passing a specific value, `ReleasedProgram` refers to programs done by any programmer. Passing a specific parameter — like `ReleasedProgram [Jimbo/producer]` — limits `ReleasedProgram` to the ones owned by a specific programmer — here `Jimbo`.

Since dependencies among code files are imposed by "uses"-relationships of programs, code dependencies are inferred data. Attributes `uses` and `compiled-to` of class `Program` determine attribute `depends-on` of class `Code`. Again, deductive rules provide the solution to specify such dependencies. A specialization of `Code!depends-on` might maintain out-dated dependencies. While a representation in terms of deductive rules and integrity constrains maintains the consistency of the repository at any time, project organization requirements may cause a shift to a less rigid approach, such as, recompilations on demand. Then, an administrator requires query support to identify programs affected by a modification and necessary recompilations.

To start with, `UncompiledProgram` and `ChangedProgram` gather programs that have been changed, but have not been compiled yet (fig. 5). Based on these query classes, `AffectedProgram` derives the set of programs which depend on a changed program directly or indirectly by a chain of "uses"-relationships. In particular, `program-affected-by-change` recursively computes the set of programs which depend on the modified program. In specifying `Affected-Program`, query class `ChangedProgram` is used like an ordinary class.

```
QueryClass UncompiledProgram isA Program with
  query_rule
    program-not-compiled-yet:
      $ not ( exists code/Code
            this.compiled-to = code and
            this!compiled-to.error-flag
                  = NoError) $
end UncompiledProgram

QueryClass ChangedProgram isA Program with
  query_rule
    program-modified-but-not-compiled:
      $ exists prog/Program
            this.successor = prog and
            prog IN UncompiledProgram $
end ChangedProgram

QueryClass AffectedProgram isA Program with
  query_rule
    program-affected-by-change:
      $ exists prog1/Program
            this.uses = prog1 and
            prog1 IN ChangedProgram
        or
        exists prog2/Program
            prog2 IN AffectedProgram and
            this.uses = prog2 $
end AffectedProgram
```

**Fig. 5:** Dynamic derivation of recompilations

In this case, the query rules and classes capture much the same information as deductive rules for dependencies. The difference lies in processing dependencies which became obsolete. The representation chosen here determines objects which has to be recompiled because of violated dependencies, i.e. the programs are assumed to be of major concern. On the other hand, one may want to describe satisfied and violated dependencies, that allows us to specify the consistency of the code of a system. In general, query classes are useful to infer objects and relationships which are out of order, in one way or another, and have to be made to fit by some kind of maintenance operations. On the other hand, derived objects and attributes, and integrity constraints on derived entities as well as relationships appear

feasible to ensure the consistency of a complex network of inter- and intra-related objects. Looking at both concepts, query classes are designed to be aware of objects which have to be maintained, and constraints and rules check for proper maintenance. However, there is no "neat" methodology yet to assist in such modeling issues and to provide assessments.

```
Individual ModularProgram isA Program with
  attribute
    import: Resource;
    export: Resource
  constraint
    proper-imports:
      $ forall res/Resources
          exists prog/Program
          this.imports = res
              ==> prog.exports = res $
  rule
    derive-used-programs:
      $ forall prog/Program res/Resource
          prog.export = res and
          this.import = res
              ==> this.uses = prog $
end ModularProgram

Attribute ModularProgram!version-of isA
  Program!version-of with
{ What represents the resource which has been changed
  by the modification. }
  attribute
    what: Resource
end ModularProgram!version-of
```

**Fig. 6:** Specialization to smart recompilation

Fig. 6 shows a refinement of `Program` representing programs which can be compiled separately. For lack of a better term, the specialized version is titled `ModularProgram`. `ModularProgram` additionally features `import` and `export` to represent resources required or used by a program. Resources visited by a modification are captured by a specialization of `version-of` for `ModularProgram`. `What` points to visited resources.

Roughly speaking, recompilations are limited to the program modified and the set of programs which actually uses this resource, i.e. programs not using a modified resource require no recompilation (cf. [33]). `AffectedModularProgram` (fig. 7) exemplifies necessary adoptions of the query rules to deal with smart recompilation.

```
QueryClass AffectedModularProgram isA
ModularProgram with
  parameter
    changedProgram: ChangedProgram
  query_rule
    modular-program-affected-by-change:
      $ exists res/Resource
            this.imports = res and
            changedProgram!version-of.what
                = res
          or
        exists prog/Program
            prog IN AffectedModularProgram
            and this.uses = prog $
end AffectedModularProgram
```

**Fig. 7:** Determining smart recompilations

## 4 Related Work

All three requirements for repository modeling pointed out in the previous sections have individually received the attention of quite a few researchers. In this section, we review some of these approaches and discuss their relationship to our work.

*Abstractional facilities* have been investigated intensively in software databases like IRIS [21]. Although flexible aggregation has been recognized as a problem [35], language and system support are largely missing.

As long as systems adhere to one data model — such as a data model to manage Ada programs or a data model to keep SADT diagrams — there is no need to investigate metaclasses. Project databases — such as ARCADIA's PMDB [26] — expand their scope by *extending* the data model, but they do not integrate data models originating from distinct domains. However, metamodeling emerges as a crucial issue, when systems with different data models are integrated. Actually, a metaclass can be conceived as the data model for specific system models. MONDRIAN [27] provides a metamodel for specifying software diagrams. Subdiagrams can be exchanged and reused among environments. Schematic discrepancies between heterogeneous data models are considered by higher order expressions ranging over views and relations [20]. Hence, metamodeling views schemes as instances.

*Interoperability* of data models has been investigated in the area of software engineering resulting in two lines of approaches: *single type models* and *universal representations* [34]. Both lines basically rely on a single set of entities and relationships which can be accessed directly by tools or can be processed after some sort of transformations. The database approach here uses a *unifying model* — Telos as ConceptBase's metamodel. The unifying model serves as data model for distinct schemes in line with approaches such as PCTE [25]. The decisive features to

assess approaches are the representational facilities. ERA-models (Entity-Relationship-Attribute), as found in PCTE or DAMOKLES [10], prevail. However, they are a step short, since they solely stick to structural properties.

*Assertional facilities* have been investigated from different perspectives. The ADELE program database uses different kinds of integrity constraints to represent consistent system configurations [2]. Each constraint is attached to an object and checked at the time the object is updated. An administrator has to organize constraints with respect to the integrity checker which eventually perverts the declarative nature of integrity constraints. Other systems that claim to stick to declarative specifications are influenced by logic programming. They use Prolog-like notations — like MARVEL [19] — to "make relationships among data fit". However, maintaining such a system has proven difficult. ARCADIA's object manager CACTIS employs the formal notions of attributed graphs to maintain inferred information [13].

For the most part, assertional properties are tackled from an operational stance. Conditions and events trigger procedures that check and update relationships. One major deficiency of such approaches is the coordination of triggers. An update may activate several triggers in parallel and cause follow-up triggers. Thus, updates may result in cascading triggers and updates. Predictions about the behavior and consequences of triggered procedures are difficult. Declarative representations with automatic trigger generation may improve precision and comprehension.

*Dynamic clustering facilities* have been studied from two perspectives. From a linguistic point of view, clustering supports inferred objects not stored explicitly. From an operational point of view, clustering is about maintaining derived objects dynamically and incrementally. The first perspective is about query concepts in general which provide declarative means to specify derived objects and structures. An idea similar to query classes has been proposed under the label of virtual classes by [1]. Although query concepts provide neat features such as "ad hoc" querying and independence from data representation, they haven't significantly carried over to software databases yet [4]. Query services might also serve as a reusable building block for tools connected to a software repository. PCTE follows this avenue yet focusing on "tables" of relationships [32]. The second perspective is view maintenance [5]. When conceiving programming or project contexts as views, actual working contexts can be maintained by view mechanisms. Similar, configuration of complex objects can be perceived as view maintenance when specifying configurations as queries. SIO goes a step further [3] considering preferences when multiple solutions apply.

## 5 Conclusions

In this paper, we presented a conceptual modeling language that serves as a data model of software repositories. *Telos* and its extension to cover dynamic clustering are operational in the *ConceptBase* prototype. Our approach integrates present data models invented for distinct applications, rather than replacing them by one coherent data model covering a broader scope. Our experiences in various applications led us to the identification of three orthogonal dimensions of requirements, illustrated in this paper by a perhaps oversimplified example: *abstractional*, *assertional* and *dynamic clustering*.

As each manufacturer is offering their own "standard" repository, we feel that such semantically more powerful language concepts are needed at least as soon as several repositories have to interoperate. Progress in database implementation techniques indicates the feasibility of an efficient commercial realization of such facilities within a relatively short time frame. Assertion compilation techniques, for example, have now reached a stage where they can reasonably compete with hand-written trigger programs, especially if the enormous effort of programming and the improved software quality is taken into account.

**Acknowledgements**

Thanks to Manfred Jeusfeld and the ConceptBase team at the Universities of Passau and Aachen, and to David Lauzon for commenting on an earlier version of this paper.

## References

1. S. Abiteboul, A. Bonner (1991). Objects and Views. *Proc. ACM-SIGMOD Conf. on Management of Data*, Denver, Co., 238-247.
2. N. Belkhatir, J. Estublier (1987). Software Management Constraints and Action Triggering in the ADELE Program Database. *Proc. 1st European Software Engineering Conf.*, Strasbourg, France, *LNCS 289*, 47-58.
3. Y. Bernard, M. Lacroix, P. Lavency, M. Vanhoedenaghe (1987). Configuration Management in an Open Environment. *Proc. 1st European Software Engineering Conf.*, Strasbourg, France, *LNCS 289*, 37-45.
4. P.A. Bernstein (1987). Database System Support for Software Engineering — An Extended Abstract. *Proc. 9th Intl. Conf. Software Engineering*, Monterey, Ca., 166-178.
5. J. Blakely, N. Coburn, P.A. Larson (1989). Updating Derived Relations — Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. on Database Systems 14*, 3, 369-400.
6. M.L. Brodie, J. Mylopoulos, J.W. Schmidt (1984). *On Conceptual Modelling*. Springer-Verlag.

7. U. Chakravarthy (1989). The HiPAC Project — Final Technical Report. Xerox AIT, Cambridge, Mass.
8. P.P.-S. Chen (1976). The Entity-Relationship Model — Toward a Unified View of Data. *ACM Trans. on Database Systems 1*, 1, 9-36.
9. P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou (1991). Software Information Base — A Server for Reuse. Report, ESPRIT project ITHACA, Forth/CRC, Iraklion, Greece.
10. K.R. Dittrich, W. Gotthard, P.C. Lockemann (1986). Damokles — A Database System for Software Engineering Environments. *Proc. Workshop on Advanced Programming Environments*, Trondheim, Norway, *LNCS 244*, 353-371.
11. U. Hahn, M. Jarke, S. Eherer, K. Kreplin (1990). CoAuthor — A Hypermedia Group Authoring Environment. In J.M. Benford, S.D. Bowers (eds.): *Studies in Computer Supported Cooperative Work*, North-Holland, 79-100.
12. U. Hahn, M. Jarke, T. Rose (1991). Teamwork Support in a Knowledge-Based Information Systems Environment. *IEEE Trans. on Software Engineering 17*, 5, 467-482.
13. S.E. Hudson, R. King (1988). The CACTIS Project: Database Support for Software Development Environments. *IEEE Trans. on Software Engineering 16*, 6, 709-719.
14. M. Jarke, Ed. (1991). *ConceptBase V3.0 User Manual*. Report MIP-9106, Universität Passau, Germany.
15. M. Jarke, J. Mylopoulos, J.W. Schmidt, Y. Vassiliou (1992). DAIDA — An Environment for Evolving Information Systems. *ACM Trans. on Information Systems 10*, 1, 1-50.
16. M. Jarke, K. Pohl (1992). Information Systems Quality and Quality Information Systems. *Proc. IFIP WG 8.2 Conf. Impact of Computer Supported Techniques on Information Systems Development*, Minneapolis, Minn.
17. M. Jeusfeld, M. Mertikas, I. Wetzel, M. Jarke, J.W. Schmidt (1990). Database Application Development as an Object Modeling Activity. *Proc. 16th Intl. Conf. Very Large Data Bases*, Brisbane, Australia, 442-454.
18. M. Jeusfeld, M. Staudt (1992). Query Optimization in Deductive Object Bases. In J.C. Freytag, D. Maier, G. Vossen (eds.): *Query Processing for Advanced Database Applications,* Morgan Kaufmann.
19. G.E. Kaiser, N.S. Barghouti, P.H. Feiler, R.W. Schwanke (1988). Database Support for Knowledge-Based Engineering Environments. *IEEE Expert 3*, 2, 18-32.
20. R. Krishnamurthy, W. Litwin, W. Kent (1991). Language Features for Interoperability of Databases with Schematic Discrepancies. *Proc. ACM-SIGMOD Conf. on Management of Data*, Denver, Co., 40-49.
21. P. Lyngback, W. Kent (1986). A Data Modeling Facility for the Design and Implementation of Information Systems. *Proc. Intl. Workshop on Object-Oriented Databases*, Pacific Grove, Ca., 6-17.
22. R.W. Matthews, W.C. McGee (1990). Data Modeling for Software Development. *IBM Systems Journal 29*, 2, 228-235.
23. B. Meyer (1985). The Software Knowledge Base. *Proc. 8th Intl. Conf. Software Engineering*, London, UK, 158-165.
24. J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis (1990). Telos: Representing Knowledge about Information Systems. *ACM Trans. on Information Systems 8*, 4, 325-362.
25. F. Oquendo, K. Berrada, F. Gallo, R. Minot, I. Thomas (1989). Version Management in the PACT Integrated Software Engineering Environment. *Proc. 2nd European Software Engineering Conf.*, Warwick, UK, 222-242.
26. M.H. Penedo (1986). Prototyping a Project Master Database for Software Engineering Environments. *Proc. 2nd Symposium on Practical Software Development Environments (SDE2)*, Palo Alto, Ca., *ACM SIGPLAN Notices 22*, 1, 1-11.
27. L.B. Protsko, P.G. Sorenson, J.P. Tremblay, D.A. Schaefer (1991). Towards the Automatic Generation of Software Diagrams. *IEEE Trans. on Software Engineering 17*, 1, 10-21.
28. B. Ramesh, V. Dhar (1992). Group Support and Change Propagation in Requirements Engineering. In M. Jarke (ed.): *Development Assistance for Database Applications*, Springer Verlag (to appear).
29. T. Rose, M. Jarke, M. Gocek, M. Maltzahn, H.W. Nissen (1991). A Decision-based Configuration Process Environment. *Software Engineering Journal 6,* 5 (*Special Issue on Software Process and its Support*), 332-346.
30. J.M. Sagawa (1990). Repository Manager Technology. *IBM Systems Journal 29*, 2, 209-227.
31. R. Sobiesiak, J. Mylopoulos (1991). A Conceptual Modelling Approach to Authoring-in-the-Large for Hypertext Documents. *Proc. Conf. Organizational Computing*, Atlanta.
32. M. Tedjini, I. Thomas, G. Benoliel, F. Gallo, R. Minot (1991). A Query Service for a Software Information System. *Proc. 4th Symposium on Software Development Environments (SDE4)*, Irvine, Ca., 238-248.
33. W.F. Tichy (1986). Smart Recompilation. *ACM Trans. on Programming Languages and Systems 8*, 3, 273-291.
34. J.C. Wileden, A.L. Wolf, W.R. Rosenblatt, P.L. Tarr (1991). Specification-level Interoperability. *Comm. ACM 34*, 5, 72-87.
35. S.H. Zdonik, P. Wegner. (1986). A Database Approach to Languages, Libraries, and Environments. *Proc. Intl. Workshop Object-Oriented Databases*, Pacific Grove, Ca.