Software Process Modeling as a Strategy for KBMS Implementation

Matthias Jarke, Manfred Jeusfeld, Thomas Rose
Fakultät für Mathematik und Informatik, Universität Passau
Postfach 2540, 8390 Passau, F.R. Germany

Abstract. Deductive and object-oriented databases should not be viewed as competitors but as two layers of abstraction (specification and implementation) within an overall knowledge base management systems (KBMS) architecture. Software process modeling is proposed as an efficient means to maintain the relationships between the two layers. A detailed account of experiences with implementing a deductive and structurally object-oriented system called ConceptBase gives preliminary evidence of the value of our proposal; ConceptBase may also serve as a basis for bootstrapping an environment for fully object-oriented databases.

1 Introduction

Several authors have observed an interesting discrepancy between the developments in deductive and object-oriented databases [YOKO89]. Deductive databases continue the tradition of the relational model by providing a simple and formally clean declarative mechanism for adding implicit information to databases. This information can be retrieved by a query language no more complicated than a relational one, and with equally clean semantics. However, despite a lot of research, only a limited number of applications were found which can actually make good use of this extension.

Conversely, object-oriented databases [BANC88] have a large number of immediate applications because of their seemingly happy marriage of data structure and operations in abstract data types, combined with the advantages of reusability and overloading gained by inheritance mechanisms. Yet, the lack of formal semantics for object-oriented databases has often been decried; in particular, operations are usually black boxes, at best characterized by input and output types, at worst leaving the user to guess from their name what they do.

This discrepancy becomes less surprising when we consider the functionality of both systems. Both add implicit information (gained by rules resp. methods) to traditional databases. However, while object-oriented databases provide full programming power using all the application experience and cleverness their programmer can muster, deductive databases are much more limited: all procedural knowledge must be specified declaratively *and* this declarative description must be handled automatically by a general deduction mechanism.

This observation is the foundation of our proposal for the implementation of systems that combine the advantages of formality in deductive databases and of general applicability and efficiency in object-oriented ones; following [BM86, ULLM89], we call such systems Knowledge Base Management Systems (KBMS). The basic idea is that **deductive and object-oriented databases should *not* be viewed as competitors but as two layers of the same system architecture** (fig. 1-1). A declarative *specification* of data, transactions, and scripts of object interaction in the style of deductive databases is *implemented* as an object-oriented database in which the transaction specifications become procedures realized in some suitable programming language.

This work was supported in part by the European Community under ESPRIT contracts 892 (DAIDA) and 3012 (COMPULOG), and by the Deutsche Forschungsgemeinschaft under contract Ja445/1-1. The authors are grateful for discussions with John Gallagher, Rainer Haidan, Manolis Koubarakis and John Mylopoulos, and for implementation work to Michael Gocek, Eva Krüger, Hans Nissen, Martin Staudt, and Thomas Wenig.

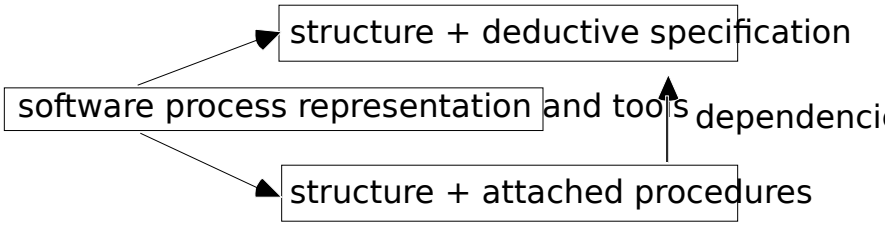

Fig. 1-1: Two-layered KBMS architecture with software process model to maintain dependencies between efficient implementation and deductive specification

1.1 Overview of the Approach

In such a KBMS architecture, ambiguities in the specification, and inefficiencies caused by its naive interpretation with a general deduction mechanism, must be removed by a *design process* that relates the specification to its implementation. Attempts to build compilers for the few semantic data models that do include procedural components (e.g., Taxis [NCL*87]) have shown that this process can be only partially automated: it involves choices that only become understood very slowly. Moreover, changes of specifications may have implications on the mapping between the two levels. For example, when entering, changing, or deleting a method specification, the system must first be able to figure out how to keep the change as local as possible with respect to the implementation layer; second, it must provide an efficient *incremental* implementation of the change with as little human activity or disturbance of onging knowledge base usage as possible.

Obviously, incremental mapping from object-oriented specifications to correct and efficient implementations is a very difficult problem. The need for specifying active DBMS components has already been recognized the late 1970´s [BR84] and there is some work on incremental compilation even in the database area [KP81]. Closely related to our task is also the work on schema evolution [BKKK87]. However, while data structure mapping is relatively well understood from a decade of research in database design, research in mapping transaction specifications to efficient code automatically has only recently attracted the attention of researchers. Only for more restricted specification languages, in particular of course the rules and constraints of deductive databases themselves, efficient mappings are known [ULLM89] although the algorithms have not been used this way; this gives us a nice way to maintain efficient structurally object-oriented deductive databases.

Besides the existence of (preferably incremental) mapping algorithms that implement deductive specifications, there is a second prerequisite for our two-layered KBMS architecture to work. It is this second prerequisite that has led to the title of this paper.

If we want to maintain consistency between the declarative and the procedural level, information about the process of mapping and remapping must be preserved. This involves knowledge about the specifications, the created data structures and procedures, the mapping decisions that were made to relate them, and the tools or algorithms used to execute these mapping decisions. A process knowledge base is needed to manage this information; it thus defines the "software object development environment" in which the individual algorithms are embedded. In the software engineering community, the term *software process model* has recently become popular for such descriptions of specification-implementation relationships[1]. The hypothesis of this paper is therefore that software process modeling may become an important strategy for KBMS implementation.

[1] Actually, software process models cover a larger portion of tasks in software engineering [OSTE87] but we shall concentrate on this specific aspect here; see, however, [JJR89] for other applications of our model.

1.2 A Case Study: ConceptBase

Over the past few years, we have been involved with the construction of a KBMS called ConceptBase which we shall use here to provide some backing to the hypothesis of this paper.ConceptBase supports the knowledge representation language CML/Telos [KMS*89] which integrates deductive rules and integrity constraints as well as an interval-based time calculus into a powerful structurally object-oriented framework with full classification, generalization, and aggregation hierarchies, and attributes as first-class objects. ConceptBase is operational on SUN and MicroVAX workstations under UNIX resp. VMS. A first prototype containing about 40.000 lines of Prolog/ SUNView code was completed in spring, 1988, and distributed to about a dozen institutions in Europe and North America for experimental applications [JJR88]. A second prototype with the functionality reported in this paper has just been finished [EJJ*89]; we are working on a third one which will handle multimedia objects and coordinate multiple users through conversation models [HJK*89].

In implementing ConceptBase, we tested the strategy proposed in this paper. First, we set up an object-oriented kernel system and defined a software process model in it; since Telos does not allow methods, we had to extend the language by introducing triggered external procedures which are invisible for the end user of the language. In ConceptBase, the kernel as well as the triggered procedures are implemented in Prolog[1], and parameter substitution follows the Prolog conventions. This implementation-level model bears some similarity to active databases, such as Postgres [SHP88] or HiPAC [DBM88][2].

The specification layer of ConceptBase offers the full Telos language. In addition to the same structural language as at the implementation layer , it includes predicative deduction rules and integrity constraints, knowledge base version and configuration support, and a usage environment that supports a hypertext style of interaction. The software process model was used to map these three components to the kernel representation, as sketched in fig. 1-2.

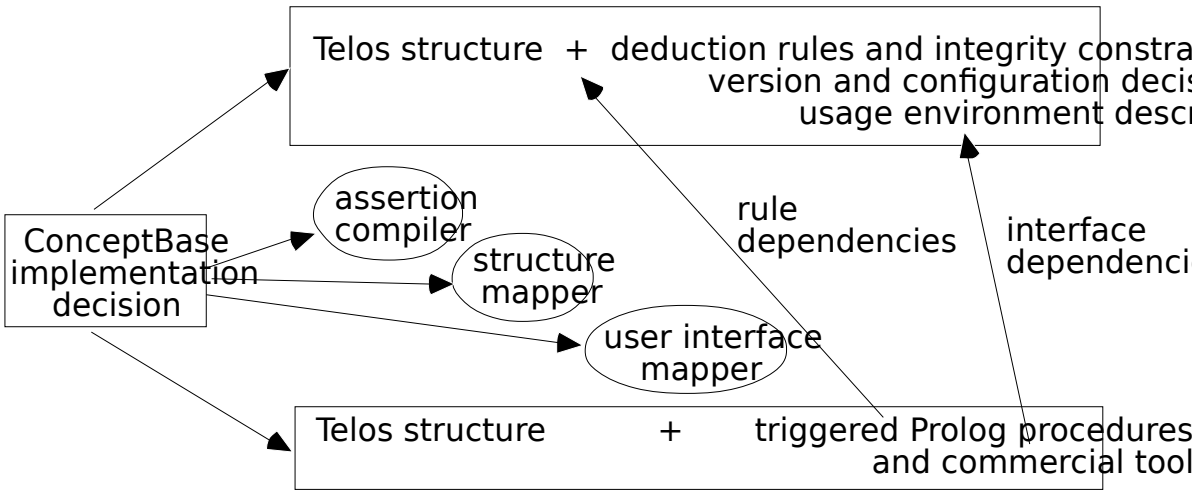

Fig. 1-2: Application of software process modeling approach in ConceptBase

The remainder of this paper is organized as follows. In section 2, we describe the implementation layer kernel of Telos and formalize the software process model in terms of this kernel. Sections 3 to 5 describe the applications shown in fig. 1-2. Section 6 outlines a more general incremental mapping environment which bootstraps from what we have so far.

---

1 together with a number of other tools which come with the BIM-Prolog system, e.g., graphics primitives.

2 The HiPAC model also comes closest to our approach in general, in that its input specifications are declarative and only the actions are programmed; a precise comparison is difficult since, to our knowledge, neither a concrete language syntax nor experiences with an implementation of HiPAC have been reported.

2 An Object-Oriented Implementation Layer: The ConceptBase Kernel

The "implementation layer" kernel of ConceptBase offers the target (and implementation) language for the mapping of higher functionalities. Except for the external procedures (called "behavior objects") which are not part of the visible language but needed for the implementation, this layer can itself be understood as consisting of a simple specification and of an implementation in a programming language (Prolog). The kernel implements a simple basic data structure, called a proposition, a few structural axioms for correct aggregation, classification, and generalization, and the two basic operations, ASK and TELL [KMS*89]. In this section, we first summarize the main features of this Telos kernel, and then define the software process model we use for the ConceptBase implementation within that language.

2.1 The Telos Object Language

The knowledge representation language Telos [KMS*89] evolved in the course of various application experiments in two ESPRIT projects from earlier work on the semantic modeling language Taxis [MBW80] and the requirements modeling language RML [GBM86] at the University of Toronto. Besides our implementation, there have been two others with different philosophies [GS86, TK89]. Telos´ structurally object-oriented framework generalizes earlier data models and knowledge representations, such as entity-relationship diagrams or semantic networks, and integrates them with temporal information and with predicative assertions. This combination of features seems to be particularly useful in software information applications such as requirements modeling or software process control. A formal description of Telos can be found in [KMS*89]. The following example shall be used to illustrate the language:

> *A company has `employees`, some of them being `managers`. Employees have a `name` and a `salary` which may change from time to time. They are assigned to `departments` which in turn are `headed` by managers. The `boss` of an employee can be derived from his department and the manager of that department. No employee is allowed to earn more money than his boss.*

The object `Employee` is declared as an instance of the system object `INDIVIDUALCLASS`, meaning that `Employee` is both a class and an individual. It has four attributes: `name`, `salary`, `dept` and `boss`. These attributes are themselves objects that link `Employee` to other objects. The class `Manager` is defined as a specialization of `Employee`. It inherits the four attributes; moreover, all instances of `Manager` are also regarded as instances of `Employee`:

```
INDIVIDUALCLASS Employee WITH
    attribute
        name: String
        salary: Money
        dept: Department
        boss: Manager
END

INDIVIDUALCLASS Manager ISA Employee

INDIVIDUALCLASS Department WITH
    attribute
        head: Manager
END
```

Telos represents two kinds of *temporal information*: the history time and the belief time [SA85]. The history time of an object states during which time interval the proposition is true. The belief time of an object is the time during which the knowledge base knew that object. The example below shows both times for an instance of the class `Employee`. The history time when `Bill` earns `20000` is `1988` but the knowledge base learned about this on `11-Jan-1989`:

```
INDIVIDUAL bill IN Employee WITH
    name
        hisname: "William T. Miller"
    salary
        earns: 20000 at 1988 believed 11-Jan-1989+
END.
```

The individual `bill` also shows another feature : attribute classes specified at the class level do not need to be instantiated at the instance level. This is the case for the `department` attribute of `Employee`. On the other hand, they may be instantiated more than once:

```
INDIVIDUAL mary IN Manager WITH
    name
        hername: "Mary Smith"
    dept
        currentdept: PR
        advises: R&D
    salary
        earns: 15000
END.
```

Telos treats all three kinds of relationships (`attribute, isa, in`) as objects; a small set of basic axioms hardcoded in the implementation defines their semantics [KMS*89]. Thus, each attribute, instantiation or generalization link of `Employee` may have its own attributes and instances; for example, each of the four `Employee` attributes is instance of an attribute class denoted by label `attribute` but can also have instances of its own. For example, the attribute with label `earns` of `mary` is an instance of attribute `salary` of class `Employee`; syntactically, attribute objects are denoted by appending the attribute label with an exclamation mark to the name of some individual. The relationship between `salary` and `earns` could be expressed as

```
ATTRIBUTE mary!earns IN Employee!salary.
```

This powerful property of Telos is enabled by the fundamental data structure of so-called *propositions* which also serve for the definition of the Telos semantics [KMS*89]:

```
propid = <sourceid, label, destinationid, validityinterval>.
```

The proposition (=object) with identifier `propid` declares that the proposition `sourceid` has a relation called `label` to `destinationid` for the time `validityinterval`. An excerpt of the propositions for our example shows how the temporal information is represented in the formalism, and stresses the uniformity of the data structure:

```
Employee = <Employee,-,Employee,Always>
p4 = <Employee,salary,Money,Always>
...
bill = <bill,-,bill,Always>
...
p100 = <bill,earns,20000,1988>
p101 = <p100,instanceof,p4,1988>
p102 = <p101,instanceof,InstanceOf,11-Jan-1989+>
...
```

The proposition representation serves as the basis for the language formalization [KMS*89] but also for the graphical display of Telos models; since proposition and frame representation are equivalent, a hypertext-like switching between textual and graphical modes of interaction is very natural (and heavily used in ConceptBase applications).
Figure 2-1 shows a subset of the objects defined so far. Unlabelled links stand for instantiation relationships, greyed links for specialization. The remaining links are attributes; though not shown in the figure, all attributes are instances of the `attribute` link of `CLASS`. The objects `CLASS`, `INDIVIDUALCLASS`, `String` and `Money` are predefined. Time intervals different from `Always` are attached to the links. The predefined class `InstanceOf` holds all instantiation links. The validity intervals of the `instanceof` links of `instanceof` links (like `p102`) are used to represent belief

times.

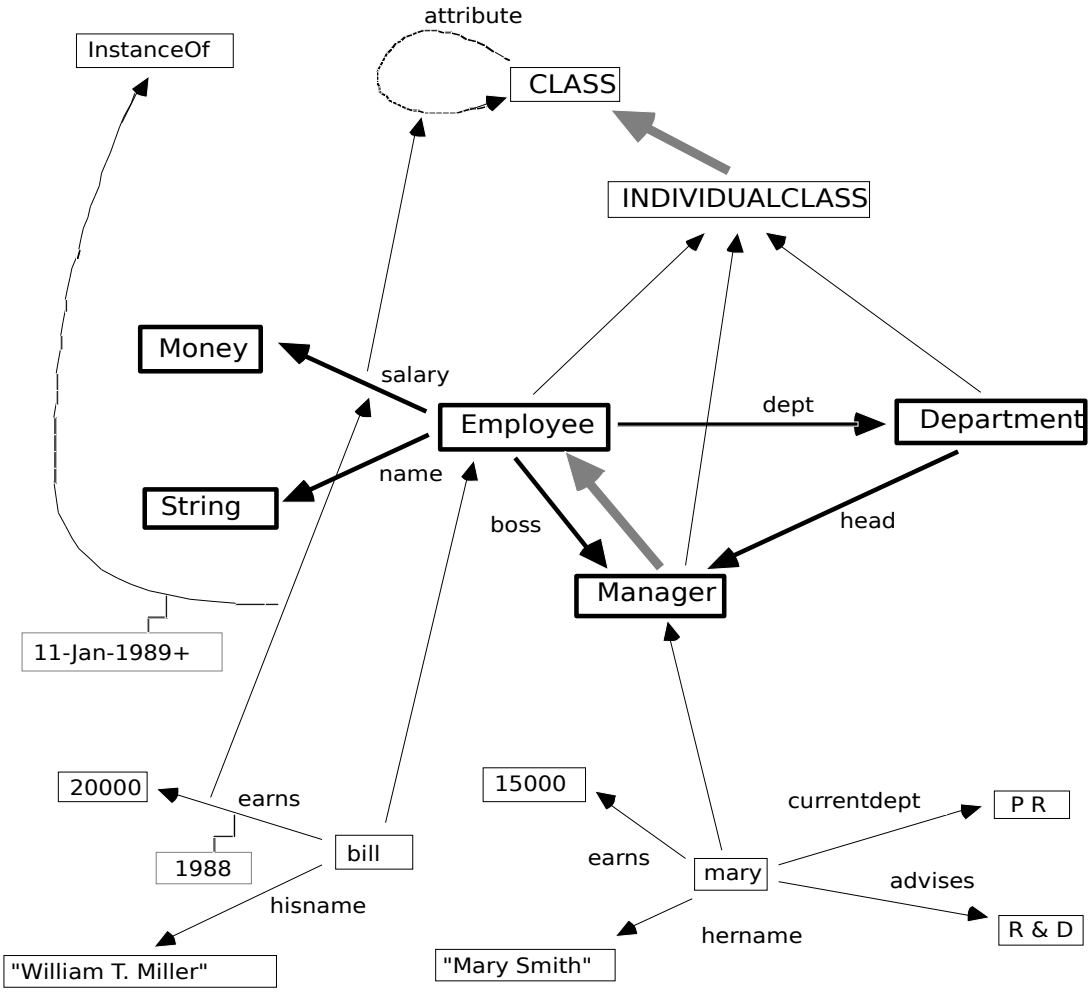

Fig. 2-1: Temporal information and instantiation in Telos´ network syntax

The user interacts with a knowledge base by operations ASK and TELL. Delete and update operations are not included directly; instead, there are operations RETELL and UNTELL which end the belief intervals of objects [KMS*89].

The actual implementation of the Telos object language in the ConceptBase kernel system is again divided into two layers [JJR88]. The upper one is responsible for managing frame structures and operations, the lower one for physical storage strategies using graph algorithms based on the network representation. We shall not discuss storage issues in this paper.

2.2 The Decision-Object-Tool (D.O.T.) Model

In this kernel language, we can now formally introduce our model of software development and maintenance processes. First, we extend the system class `CLASS` of fig. 2-1 by built-in attribute categories for dependencies and triggers, and then we introduce the metaclasses that define the structure of our model: intermediate results or design *objects*, design *decisions* that create or alter the objects, and design *tools* that support the execution of decisions.
The basic system object `CLASS` is extended by two additional attribute classes. One induces a special kind of attributes called `dependson`. The other introduces a class of `trigger` links to externally defined procedures. With these extensions, `CLASS` is defined as follows:

```
INDIVIDUALCLASS CLASS WITH
     attribute
         attribute : CLASS
         dependson : CLASS
         trigger : BehaviorObject
END
```

Given the basic axioms of the Telos object language, this says that classes (= instances of `CLASS`) may have attribute classes of categories `dependson` and `trigger` which in turn lead to other classes respectively executable procedures. The (built-in) semantics of this is as follows. Classes which have attribute classes of type `dependson` contain instances that are derived objects; the dependency attributes relate these objects to the objects they were derived from. Attribute classes of type `trigger` are associated with an activation condition (e.g., a new instance of the class is created, or an instance link is cut); an instance of the procedure they point to is generated and executed when the invocation pattern becomes true [KDM88].

Now let us take a closer look at the basic approach shown in fig. 1-1 of section 1 which is repeated in fig. 2-2 for convenience.

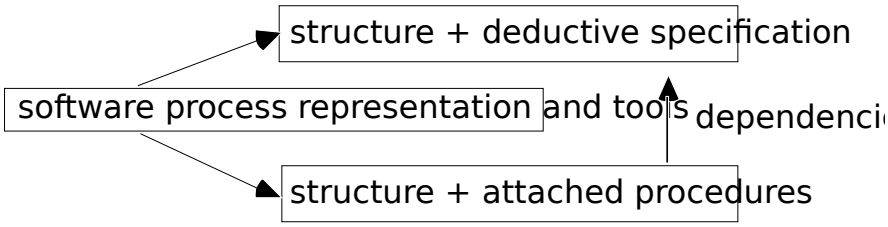

Fig. 2-2: Two-layered KBMS architecture connected by software process model

At first, assume that both the inputs and the outputs of the software process (i.e., the specification and the implementation) are black box objects. We only know that these two objects exist. We also assume that an arbitrary semantic description in Telos can be attached to an object, and that the content of the black box (which need not be understandable in Telos) is pointed to by some external reference. We call objects so described *design objects* and define the following metaclass for them:

```
INDIVIDUALCLASS DesignObject IN MetametaClass WITH
     attribute
         objsemantic : CLASS
         objsource : ExternalReference
END
```

Now consider the rounded box in fig. 2-2. There are usually many possibilities how to implement a specification; *design decisions* are required to choose among these possibilities, based on general mapping knowledge and specific situation knowledge. The whole software process can be viewed as a large design decision which is composed from many smaller ones; execution of these small sub-decisions creates *dependencies* among their corresponding input and output components; taken together, these dependencies form a semantic description of the global decision. At the same time, they can also be used to identify subdecisions that must be replayed when an incremental change is intended. The following metaclass definition captures this intuition:

```
INDIVIDUALCLASS DesignDecision ISA DesignObject WITH
```

```
    attribute
        from, to : DesignObject
        decsemantic : DecisionDescription
END

INDIVIDUALCLASS DecisionDescription IN MetametaClass WITH
    attribute
        dependencies : CLASS!dependson
END
```

The above metaclass definitions are very general in that the design objects mentioned there can be arbitrary intermediate results of any development process. Our intuition behind fig. 2-2 is stronger. Firstly, we expect that the implementation-level system is actually executable, not just an intermediate design result. Secondly, we expect that the deductive method specification actually expresses a relationship between input and output information. In other words, the specification itself has the structure of a design decision. Our two-layered KBMS (or small part thereof) should consist of a specification which follows the structure of the `DesignDecision` metaclass, and an implementation which is a `BehaviorObject` in the sense discussed earlier. This is captured by the notion of `DesignTool`:

```
INDIVIDUALCLASS DesignTool ISA DesignDecision WITH
    attribute
        from : DesignDecision
        to : BehaviorObject
END
```

We could call this metaclass "conceptual software object" or "reusable component" since it captures what a subsystem does, and how its specification is related to its implementation. The name `DesignTool` is justified if we apply the above definition not to the deductive specification of the method but to the mapping task itself. For example, the "assertion compiler" in fig. 1-2, takes as its specification a methodology how to implement assertions as triggered procedures, and as its implementation a corresponding procedure. In this case, the dependencies show what has to happen if assertions are added, deleted, or altered; or if the optimization algorithm implemented by the compiler is changed.

Fig. 2-3 represents the semantic network view of this software process model, emphasizing the relationships between objects, decisions, and tools. From the design perspective, the decision-object-tool model describes where the behaviors come from and what they do. At execution time, only the trigger attributes attached to object descriptions (`CLASS`) are needed, and the design information is needed only when change occurs.

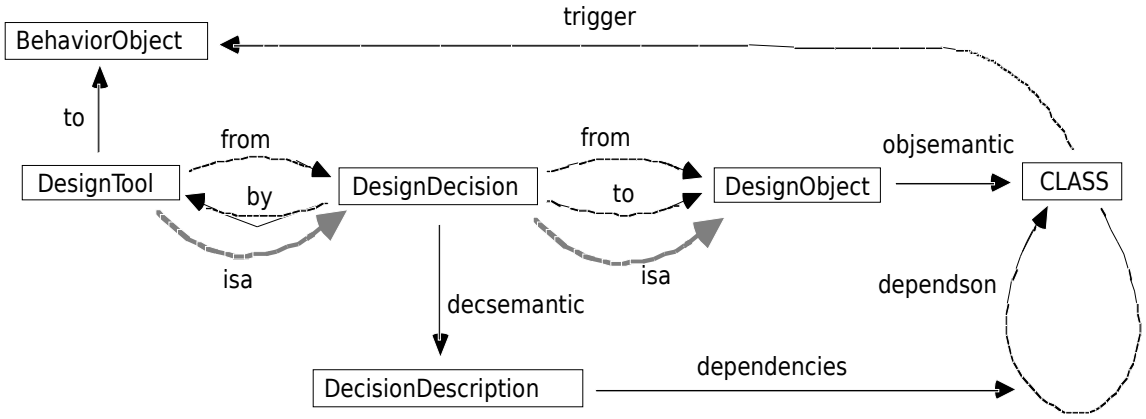

Fig. 2-3: Software process metamodel formalized in Telos kernel language

So far, we have only outlined the basic structure of our model at a very abstract level, that of metametaclasses. This model can be instantiated with the description of a particular software environment, consisting of specific classes of design objects, design decisions, and tools; in our case, this level corresponds to the description of ConceptBase as sketched in fig. 1-2 and elaborated in the next three sections. This environment model can be instantiated again with a set of particular specifications (such as rules about `Employee`), and yet another time to apply them to

actual data (such as those about Bill and Mary).

3 Rules and Constraints as Knowledge Base Objects

We now turn to the first example of how the software process model and the structural kernel described in section 3 can be exploited for developing efficient implementations. In order to integrate predicative assertions into the object-oriented kernel, we represent both the assertions themselves and the way how assertion evaluators are attached to objects with the D.O.T. model. The assertion model we obtain is similar to the one proposed for HiPAC [DBM88] and its internal representation lends itself well to the large number of graph-based algorithms for deductive query processing and integrity checking in the literature [ULLM89]. Interesting features we obtain as side benefits of our approach include the following:

- In contrast to most other graph-based implementations, the structures needed for efficient implementation are objects of the language (Telos) itself, rather than obscurely hidden in implementation code.
- Modeling the development process for rule evaluators in the same manner as the rules themselves gives us a way to integrate different query optimization and integrity checking algorithms gracefully in order to obtain an overall efficient control strategy. It provides an alternative way of structuring extensible DBMS toolkits [FREY87, GD87] and allows incremental addition/ deletion of assertions.
- The possibility to represent derived data and dependencies in the model provides a basis for maintaining redundant information. Applied to the assertions themselves, we can materialize selectively rule-derived views to make, e.g., integrity control more efficient [BLAK87]. Applied to the design process, it generates bases of triggered procedures already at assertion insertion time. For example, we have implemented a design-time time version of the deductive integrity checking algorithm proposed by [BDM88] to save substantial work during system usage.

We first describe the representation and evaluation of assertion objects, and then only briefly sketch the design process model since it should be pretty obvious at this point.

At the language surface, predicative assertions are integrated in Telos as special attribute values of metaclass `Assertion` which have no further structure known to the user. Their role as either deduction rules or integrity constraints is defined by the way how assertion objects are attached to other objects by attribute links. In principle, several different assertion languages can be integrated in this fashion and this fact is used in ConceptBase to integrate special-purpose theorem-provers for software engineering. Here, we focus on a first-order language as in deductive databases [KMS*89]. In our example, consider the rule that deduces an employee´s boss, and the constraint that no employee should earn more than his boss:

```
RETELL Employee WITH
    rule
        bossrule: $ forall e/Employee,d/Department,m/Manager
                    e.dept=d and d.head=m ==> e.boss=m $
    constraint
        salarybound: $ forall e/Employee,m/Manager,x/Money,y/Money
                       e.boss=m and e.salary=x and m.salary=y
                       ==> x =< y $
END
```

Thus, if `bill` has the department `PR` which is headed by `mary`, then `mary` is the boss of `bill` and `bill´s` salary should not be higher than `mary´s`.

In terms of Telos´ structural representation, we can view the rule text as labeling an individual proposition such as

```
rule1 = < rule1, $forall e/Employee ...$, rule1, time1 >
```

where `time1` determines during which interval the rule should be applicable.

The class `Assertion` can itself be viewed as a specialization of the metametaclass `DesignDecision` of the software process model. For rules, the conditions correspond to the `from` design objects, and the conclusion to the `to` design objects; for constraints, the `to` object is a special object class which can take values ´consistent´ or ´inconsistent´. Being a design decision, each assertion can be associated with a set of tools called assertion evaluators. Moreover, each input object mentioned in the assertion´s literals can have a trigger attribute that fires these evaluators under certain conditions (e.g., when it is instantiated).

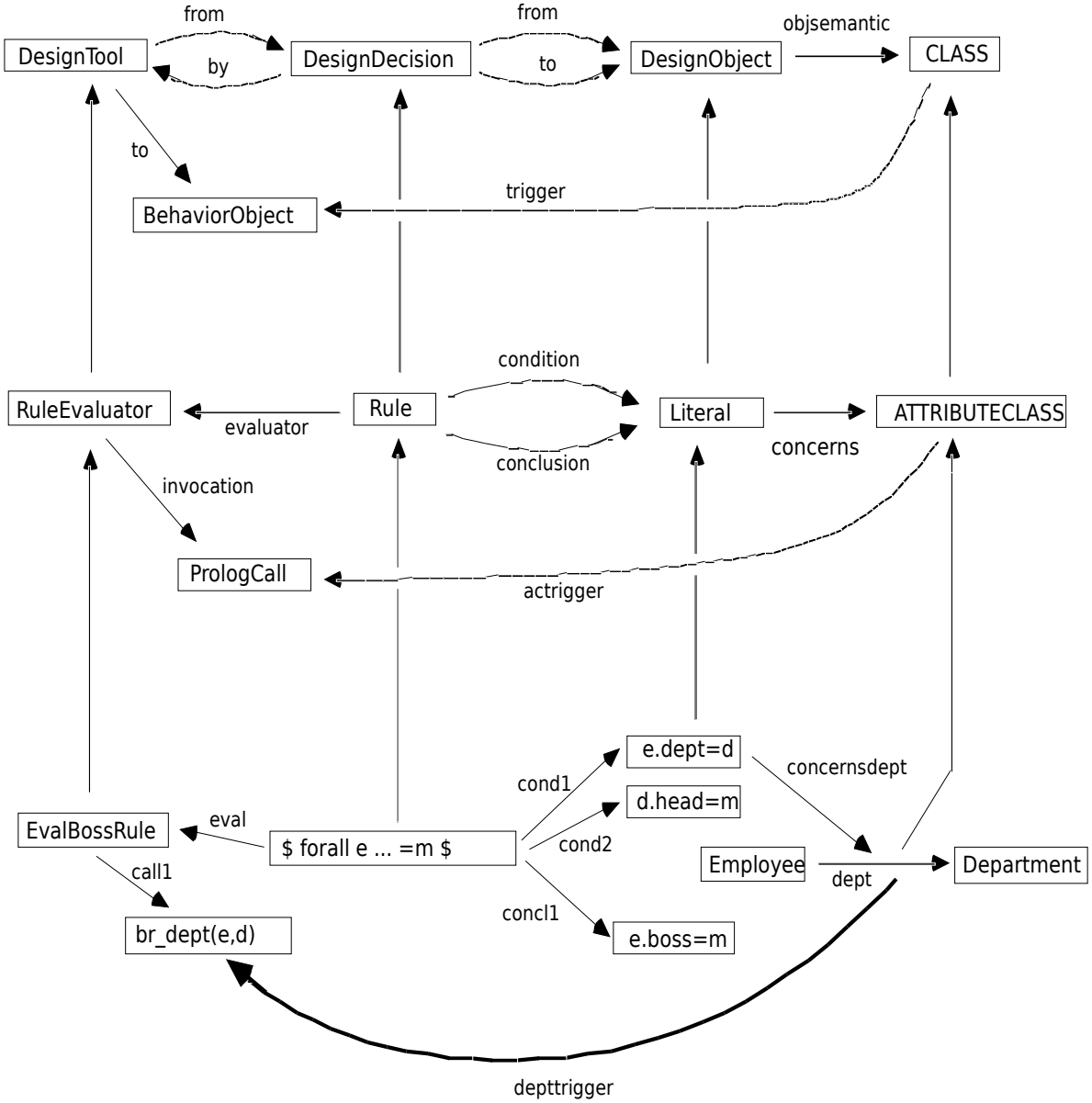

Fig. 3-1: Modeling assertion languages as decision classes

Figures 3-1 and 3-2 illustrate the model. Figure 3-1 shows part of the model for assertions as an instantiation of the generic software process models. As a particular instance of this assertion model, the structure of the example deduction rule is also sketched. This shows the situation at compile time: the rule is entered as an instance of metaclass `Rule` such that its input description points, e.g., to the `dept` link.

If there are more rules and constraints, an object such as the `dept` link may be associated with a large number of triggers, some to be fired upon instantiation, some upon invalidation of an instance link, etc. Thus, we have an object with encapsulated procedures; but of course, the existence of these procedures depends on the existence of the assertions that they implement, and on the further acceptance of the implementation algorithm (here, a forward chaining technique proposed by Decker [BDM88]).

Figure 3-2 illustrates the firing of such generated triggers for the deductive integrity optimization algorithms proposed in [BDM88] of which an incremental design-time version was implemented in ConceptBase [KRÜG89]. The algorithm generates specialized procedures separately for insertion and invalidation of each object where this operation could violate the constraint. Forward evaluation procedures are attached to rules whose consequents generate or delete objects such that a constraint could be violated. In this way, only the necessary forward operations are conducted for each update, assuming that he KB was consistent before. Special tests have been added by us for the addition or deletion of rules and constraints where data already exist.

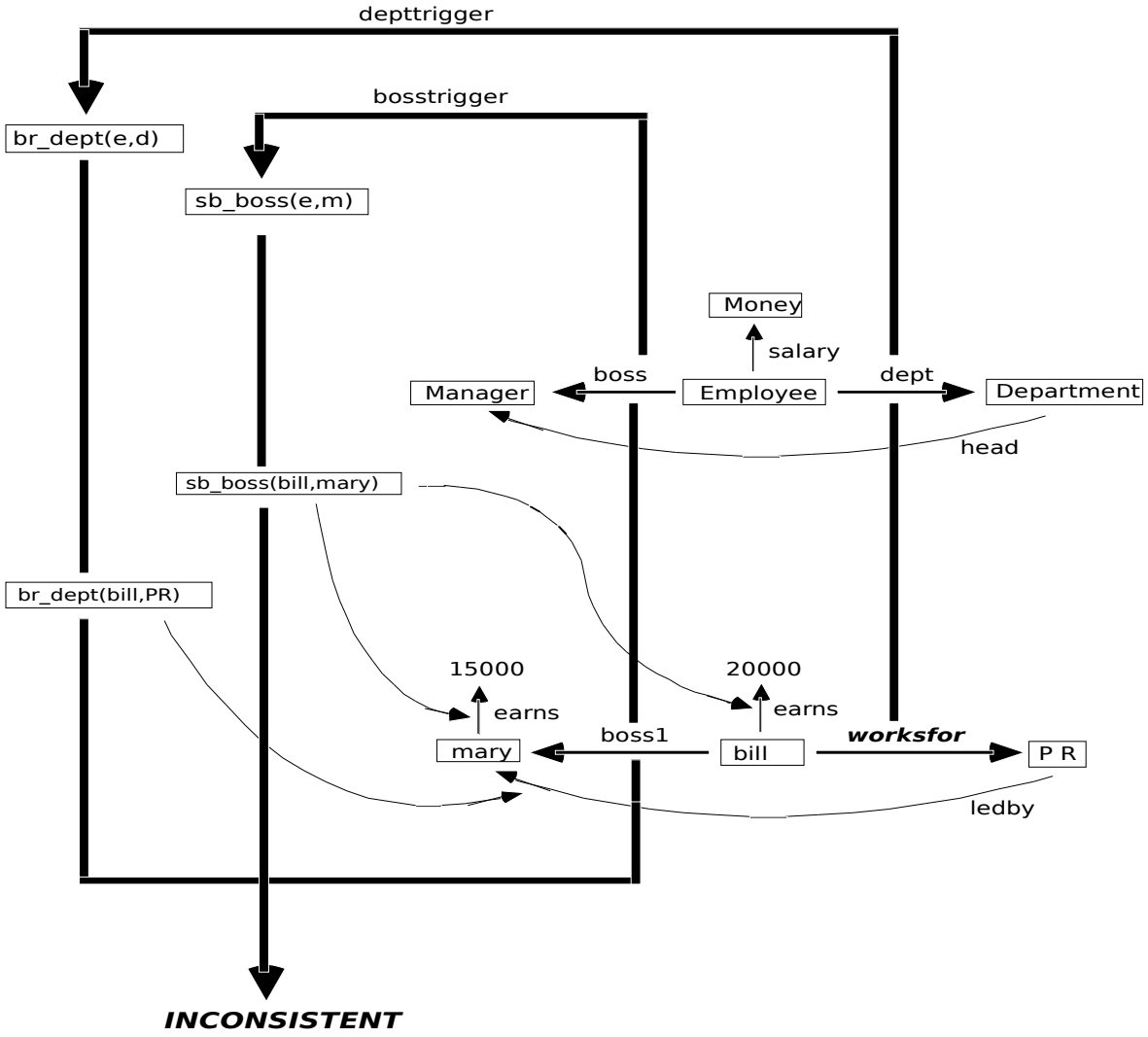

Fig . 3-2: Using the network for deductive integrity checking

When we insert the `worksfor` attribute of object `bill` as an instance of the `dept` attribute of `Employee`, the procedure `br_dept(e,d)` is triggered. The formal parameters are replaced yielding the actual call `br_dept(bill,PR)` which can be considered an instance of the procedure. Using the `ledby` link of `PR` the `boss` of `bill` is computed and inserted in the KB. Since its class, the `boss` attribute of `Employee`, has a trigger, too, another procedure call `sb_boss(bill,mary)` occurs. The two salaries of `bill` and `mary` are compared and inconsistency is determined. Note that the representation of the rule object itself is completely ignored in this cycle; it is reactivated if the rule or the optimization algorithms change. The following ConceptBase screendump shows the same situation. In the graph editor window, the specific metaclasses for the algorithm of [BDM88] are shown; one editor window shows the class definition of `Employee`, another one the attempted instantiation with a new employee, `bill`, who violates the constraint as explained in the error

window.

Fig. 3-3: ConceptBase screendump demonstrating deductive integrity control

4 Knowledge Base Version and Configuration Management

The second application of this implementation strategy concerns version and configuration management. To limit search space, large-scale knowledge bases have to be partitioned. Partitioning needs to be done by topic as well as by temporal validity of the knowledge. It requires *configuration* management to compose knowledge base views from existing components. *Versioning* is intended to provide a temporal or organizational classification.
Thus, each ConceptBase knowledge base is represented by a (possibly recursive) configuration of subparts representing content-oriented modules and temporal versions. Following the same line of argument as before, this section discusses how to model versions and configurations conceptually by design decisions, and how to model the process of implementing these conceptual decisions as structures and operations on existing software version and configuration management tools which work at the file level. In this paper, we apply this model to ConceptBase itself; in [RJ89], it is elaborated in much more detail for general version and configuration management tasks in software engineering.

4.1 Conceptual Version and Configuration Decisions

In a first step of applying our software process model to this problem, we could view configurations as a special kind of design decisions similar to deduction rules. In this case, configurations would be created on demand from these rules and forgotten immediately afterwards. Of course, this would be rather inefficient. Therefore, we enrich the rule model by allowing redundant storage of the decision results. In this case, correspondences (possibly many-to-many) between multiple redundant representations of the same data must be maintained to ensure their mutual consistency; dependencies enable us to describe these correspondences at a very detailed level, facilitating incremental change without total recompilation along the lines of systems such as Cactis [HK87]. At the same time, the model is quite compatible with version/configuration/equivalence modells as proposed in [KCB86].

The basic concept to structure a knowledge base is a *module*. A module represents a view of objects of the knowledge base and comprises two properties: an interface and an implementation. The interface describes properties visible from the outside of the module. The implementation represents a configuration of objects satisfying the interface description. Each configuration object is justified by a configuration decision which takes the components as its inputs and the module interface as its output. In the following, we discuss two kinds of configuration and their correspondences in ConceptBase, namely conceptual configurations (which objects belong together from a content perspective?) and source configurations (which objects are stored together physically?).

Figure 4-1 shows two versions of a conceptual module `Telos_CompanyModel` representing the complete Telos model of a company as presented in section 2.1. The decision `Configure-Company` configures a first version of this object from the design objects `Telos_Department`, `Telos_Manager` and `Telos_Employee86`. These design objects are individuals of the process model and used to represent the Telos objects `Department`, `Manager` and `Employee`. `Telos_Employee86` represents the individual class `Employee` before introducing assertions. The decision description `ConfigDescr` aggregates the semantic dependencies of this configuration which only refer to a temporal constraint on the configuration (grey vertical arrows in figure 4-1), namely that all components of the configuration should be valid during 1986. Consistency and maintenance of configurations can be handled by such dependencies.

Assume that the rules mentioned in section 3 are added to `Employee` in 1988. The new *version* of `Employee` is named `Telos_Employee88`. `Telos_Employee88` is not allowed to be part of the configuration `ConfigureCompany` because of the temporal dependency. `ConfigureCompany88` represents the new version of the configuration decision.

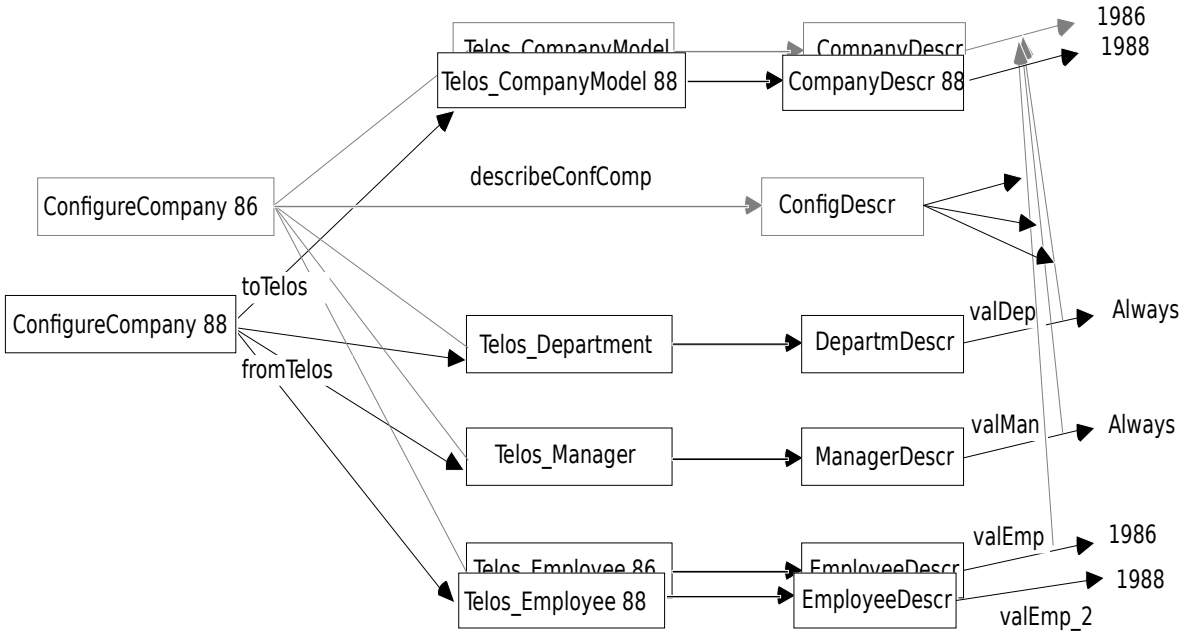

Fig. 4-1: Configuration of the extended company model

4.2 Representing the Implementation-Level System

The storage of the company model on external devices, i.e. the implementation of the conceptual configuration, is depicted in figure 4-2. The semantic description of source code design objects may comprise their allocation, version identifiers, access rights, etc. Similar to conceptual modules, source codes can be configured to source modules. For instance, a group in a source code management system like CMS [DEC82] is a set of sources with an associated access specification. Thus, the decision `CompSourceModule` represents the configuration of source module `CMS_CompGroup`. Solid lines in figure 4-2 show the source configuration for the 1986 version of the company model. In the greyed new version, `CompSourceModul 88`, the developer has decided to keep the source object `CompanySources` unchanged but to store the assertions in an extra file `CompanyAssertions` intended to collect all constraints and deduction rules of the model. Thus, the new version of the source model consists of a source object with the old source text and an additional file.

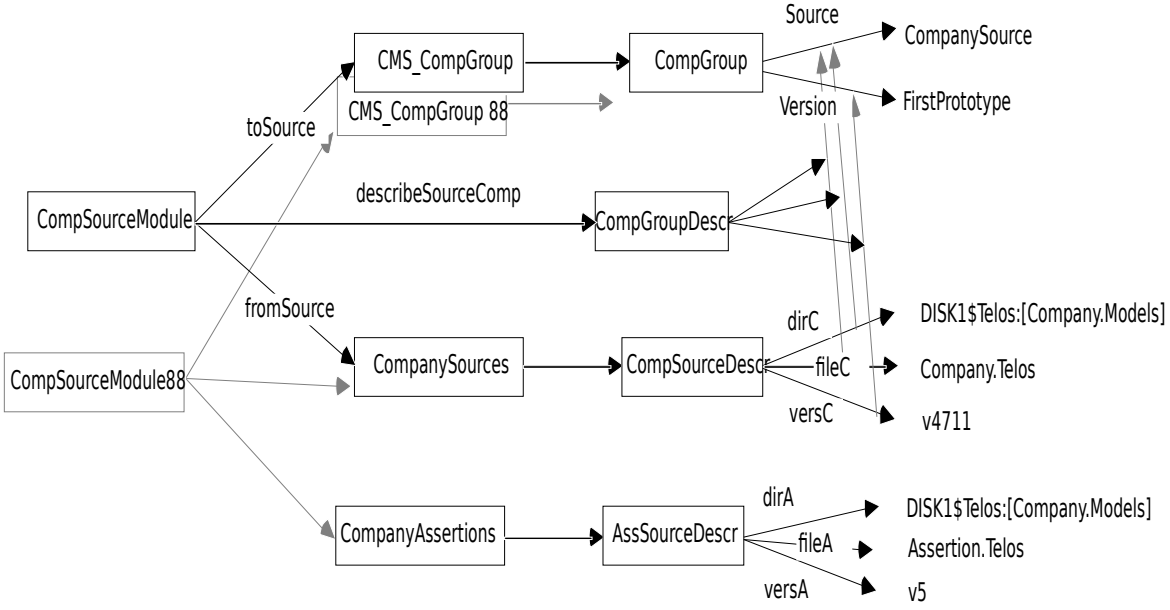

Fig. 4-2: Configuration of source objects for the company model

4.3 The Conceptual-To-Physical Mapping

Finally, we look at the correspondences between the conceptual and the source view of the knowledge base, i.e. implement*ing* conceptual configurations by existing implementations. This implementation process is represented by design decisions (details about process support and interactive decision assistance in [RJ89]) which map a conceptual configuration of Telos classes to a reasonable source representation and source configuration.

Figure 4-3 shows the correspondence between the modules `Telos_CompanyModel` and `CompSourceModule`. The semantic descriptions `BodyDescr` and `GroupDescr` have as attriubutes the members of the module configurations. These memberships are derived from the configuration decisions `ConfigureCompany88` and `CompSourceModule88`. The semantic description of such a correspondence decision describes the interrelationships among conceptual and source memberships (grey arrows in figure 4-3), as discussed above. It can be easily seen how this kind of model could be used to determine which source modules to look at to find information about a conceptual module for a given validity interval; for instance, if the user is interested in the 1986 version, he has to retrieve only one source module.

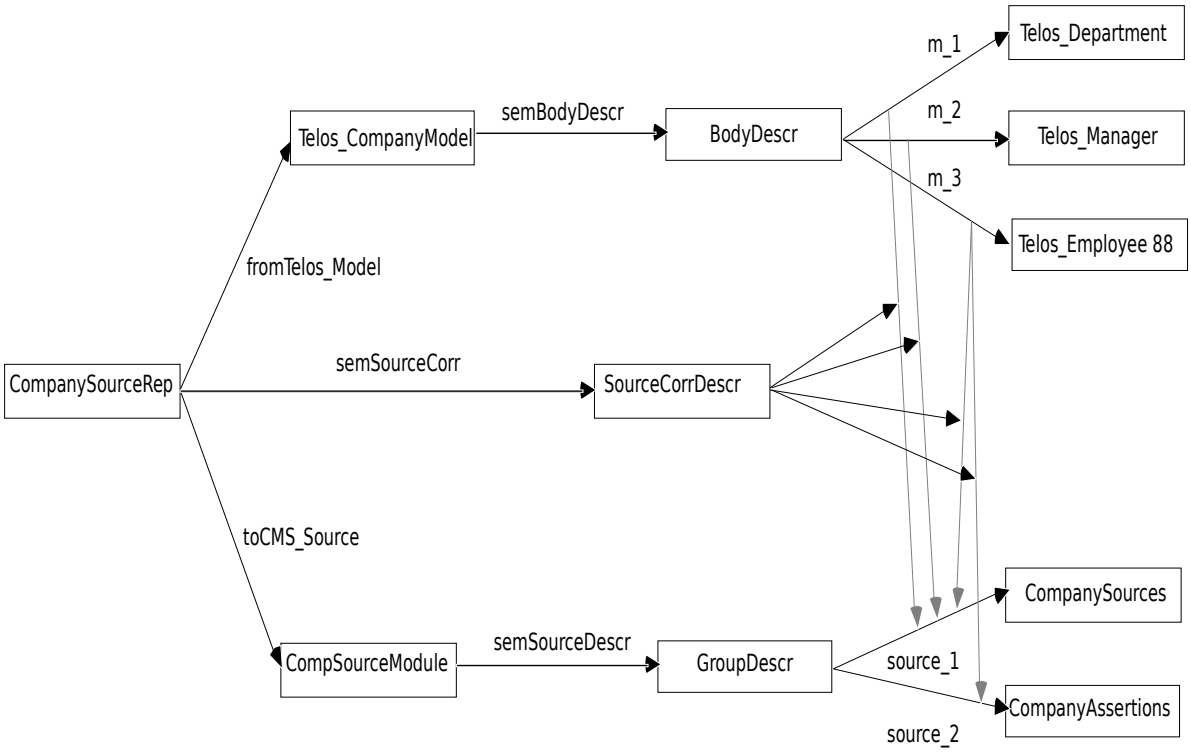

Fig. 4-3: Correspondence between conceptual model and its source representation

ConceptBase currently employs SCCS/NSE for the SUN-UNIX environment and CMS/MMS for the VAX-VMS environment for source code management; distributed source code management becomes possible due to the common conceptual configuration model. The ConceptBase screendump in fig. 4-4 illustrates this with a larger example: a knowledge base about the ConceptBase implementation we use to control our own development efforts. The screendump shows in a hierarchical browser on the SUN (lower right window) a Telos model of the CMS storage of the DEC version of the ConceptBase implementation. The right upper window shows actual interaction with this version via remote access to a MicroVAX, whereas the graphical editor on the left documents the decision instance to put the ConceptBase version on DEC together in the manner required for this situation (that is, with an ASCII terminal interface since we cannot emulate DEC graphics on the SUN).

Fig. 4-4: ConceptBase screendump demonstrating configuration management

5 User Interfaces as Configurations of Derived Data

A discussion on usage environments has to cover many topics. A first one concerns the data integration of usage environment and kernel system, i.e. how to control the consistency among knowledge base objects and graphical structures displayed on screen. A second topic concerns the functionality of tools, i.e. one has to determine a graphical presentation and functionality which supports the representational framework of the knowledge representation language. At first, this section describes a conceptual approach to represent user interfaces and control their consistency. Subsequently, it sketches the actual interface tools of ConceptBase; other interface-related issues such as dialog control can be modeled again with nested design decisions but this is ongoing work which will be reported elsewhere.

Our configuration model is almost directly applicable to interface management as well. In general, a screen consists of a set of windows each managed by a tool (e.g. an editor/ browser). Since each window handles a set of objects derived by some deduction rules, it can be represented as a configuration module (fig. 5-1) whose interface specifies what objects configure it and how to present them on the screen. For instance, `BrowserConf` describes the configuration of those objects to be displayed by a browser and `EditorConf` describes the configuration of the editor; consistency of externally displayed views can be controlled in case of modifications by the kernel system or an interface tool. Correspondences between different views also allow the propagation of constraints across windows.

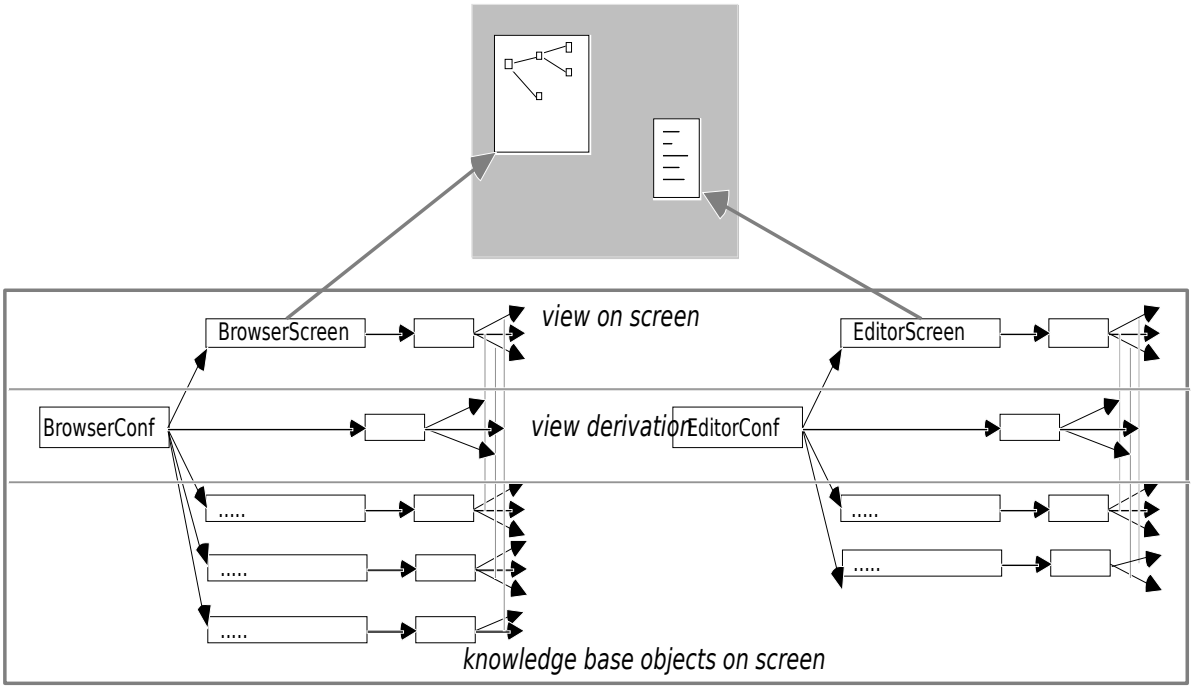

Fig. 5-1: A screen as a configuration of materialized configurations

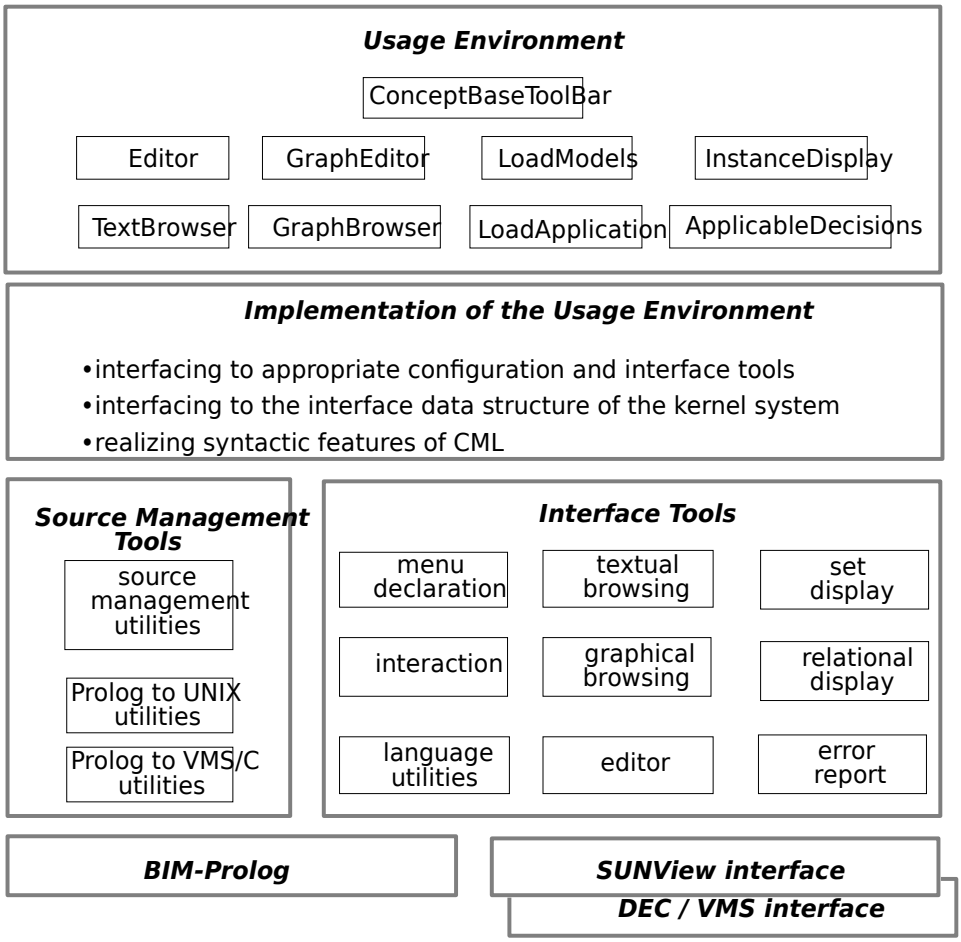

Fig. 5-2: Implementation overview of ConceptBase usage environment

To exploit the hybrid nature of Telos, we decided to define views that allow for a hypertext-like style of interaction. Therefore, the environment allows arbitrary switching between graphical and textual (frame) modes of display and interaction by attaching a selection and modification facility to each view. An impression of the interface can be gained from the screendumps of figures 3-3 and 4-4. Rather than explaining each tool in detail [EJJ*89], we just give an architectural overview.

The ConceptBase usage environment is organized in three layers, as shown in figure 5-2. The bottom layer provides a box of interface and configuration tools which process uninterpreted strings (e.g. object identifiers) and structures. These tools implement decision classes on graphical presentations and may be utilized for different purposes by the usage environment. The usage environment itself comprises tools for editing Telos objects, browsing classification or generalization hierarchies, etc. These tools support decisions which can be applied to knowledge bases. They consist of three parts. The first obtains a view to be displayed by calling a tool to manage user interaction by panels. The second derives the specified view. The third transforms this view to a structure processable by the interface objects of the tool box. The middle layer of fig. 5-2 relates screen-oriented and knowledge base-oriented views by defining correspondences between windows (screen configurations) and conceptual configurations (e.g., query results to be displayed).

6 Conclusion: Towards Incremental Object Development in KBMS

We tried to show how a decision-object-tool model of software processes can represent and support (a) the specification of implicit knowledge, and (b) the implementation of such specifications by object-oriented databases. Although our experience with ConceptBase has shown the usefulness of this approach only for relatively simple cases, we believe that it can be extended to more general cases.

Before pointing out our plans in this directions, we briefly summarize the results of this paper concerning the three different mapping tasks identified in the ConceptBase implementation.

First, we modeled deduction rules and integrity constraints as deterministic (and therefore automatable) "design decisions" and showed that the graph structures derived from such a model are identical to those used by various optimization algorithms. Moreover, the approach led us to develop incremental versions of these algorithms applicable at design time rather than system usage time.

Second, we attacked a more complicated case in which manual and automated decision-making interact, namely the creation and administration of consistent versions of configurations. This allows the user to talk about conceptual components of the system while internally using efficient commercial configuration managers which work on source objects not isomorphic to the conceptual ones.

Finally, we combined deduction and configuration aspects in modeling the ConceptBase usage environment as a configuration of derived data, similar to the approach taken in Postgres [SHP88]. This approach is proving very helpful in extending the system to true hypertext (or even hypermedia) capabilities.

To achieve more generality, one has to look for more powerful mapping technology. Work in "automatic programming" such as exemplified by the CHI/REFINE system [SKW85] is an important source for such methods; formal transaction verification techniques such as those investigated by [SS86] can also be of use. Then we can use a basic software process manager such as the current version of ConceptBase to manage and document the mapping processes supported by such tools.

In an ESPRIT project called DAIDA [DAIDA89], an initial effort was made to map general transaction specifications expressed in a purely declarative version of the semantic modeling language, Taxis [MBW80], to procedural code written in the database programming language, DBPL [BMSW89]. Taxis transactions are specified by preconditions, goals, and invariants. A theorem-proving assistant is then used to convert this specification into a set-oriented formalism and to apply verified refinement steps until a specification is reached that can be directly translated into a satisfactory database program. The design decisions and proofs are recorded by ConceptBase using an instance of the D.O.T. model, in order to reduce the need for re-proving and to maintain the relationships between specification and implementation [JJR*89]. There is still a lot of research needed for a sufficient automation of the proving process, and incrementality of the mapping algorithms as well as efficient dependency tracking procedures in ConceptBase itself remain to be investigated. Nevertheless, the initial results obtained in coupling such a transformational software development tool with a knowledge base supporting our software process model look quite promising.

A second area of further research we are interested in concerns a more direct logical formalization of our model; currently, this formalization is given only indirectly via the logical semantics of Telos. It would be quite interesting to understand the relationship of our approach to recent work on metalevel logic programming [LLOY89] which follows similar goals but also to studies of declarative update languages [ABIT88].

References

[ABIT88] Abiteboul, S. (1988). Updates: the new frontier. *Proc. 2nd Intl. Conf. Database Theory*, Bruges, Belgium, 1-18.

[BANC88] Bancilhon, F. (1988). Object-oriented database systems *Proc. ACM 7th Symp. Principles of Database Systems*, Austin, Tx, 152-162.

[BDM88] Bry, F., Decker, H., Manthey, R. (1988). A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. *Proc. EDBT '88*, Venice, Italy, 488-505.

[BLAK87] Blakeley, J.A. (1987). Updating materialized database views. Ph.D. thesis, Dept. Computer Science, University of Waterloo, Ont.

[BKKK87] Banerjee, J., Kim, W., Kim, H.-J., Korth, H.F. (1987). Semantics and implementation of schema evolution in object-oriented databases. *Proc. ACM- SIGMOD Conf.*, San Francisco, Ca, 311-322.

[BM86] Brodie, M.L., Mylopoulos, J., eds. (1986). *On Knowledge Base Management Systems*. New York: Springer-Verlag.

[BMSW89] Borgida, A., Mylopoulos, J., Schmidt, J.W., Wetzel, I. (1989). Support for data-intensive applications: conceptual design and software development. To appear in *Proc. 2nd Workshop Database Programming Languages*, Portland, Or.

[BR84] Brodie, M.L., Ridjanovic, D. (1984). On the design and specification of database transactions.In Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (eds.): *On Conceptual Modeling*, New York: Springer-Verlag, 277-306.

[DAIDA89] Jarke,M.,DAIDA Team (1989). The DAIDA demonstrator.*ESPRIT89*, Brussels.

[DBM88] Dayal, U., Buchmann, A., McCarthy, D.R. (1988). Rules are objects too: a knowledge model for active, object-oriented database systems. In Dittrich, K. (ed.): *Advances in Object-Oriented Databases*, Springer-Verlag, 129-143.

[DEC82] Code Management System: User Manual (1982). *Digital Equipment Corporation.*

[EJJ*89] Eherer, S., Jarke, M., Jeusfeld, M., Miethsam, A., Rose, T. (1989). A global KBMS for database software evolution: ConceptBase 2.0 user manual. Report, Universität Passau, W. Germany.

[FREY87] Freytag, J.C. (1987). A rule-based view of query optimization. *Proc. ACM- SIGMOD Conf.*, San Francisco, Ca, 173-180.

[GBM86] Greenspan, S., Borgida, A., Mylopoulos, J. (1986). A requirements modelling language and its logic, in Brodie, M.L. Mylopoulos, J. (eds.): *On Knowledge Base Management Systems*, New York, Springer-Verlag, 471-502.

[GD87] Graefe, G., DeWitt, D.J. (1987). The EXODUS query optimizer generator. *Proc. ACM-SIGMOD Conf.*, San Francisco, Ca, 160-172.

[GS86] Gallagher, J., Solomon, L. (1986). CML Support System. Report, ESPRIT Project 107 (LOKI), SCS Hamburg, W. Germany.

[HJK*89] Hahn, U., Jarke, M., Kreplin, K., Farusi, M., Pimpinelli, F. (1989). CoAUTHOR: a hypermedia group authoring environment. *Proc. European Conf. Computer-Supported Cooperative Work*, Gatwick, UK.

[HK87] Hudson, S.E., King, R. (1987). Object-oriented database support for software environments. *Proc. ACM-SIGMOD Conf.*, San Francisco, Ca, 491-503.

[JJR88] Jarke, M., Jeusfeld, M., Rose, T. (1988). A KBMS for database software evolution: documentation of first ConceptBase prototype. Report MIP-8819, Universität Passau, West Germany.

[JJR89] Jarke, M., Jeusfeld, M., Rose, T. (1989). A software process data model for knowledge engineering in information systems. *Information Systems 14*, 3.

[JJR*89] Jarke, M., Jeusfeld, M., Rose, T., Mylopoulos, J., Schmidt, J.W., Wetzel, I., Ziegler, A. (1989). Data and process management in formal software development. Working paper, Universität Passau (submitted for publication).

[KCB86] Katz, R., Chang, E., Bhateja, R. (1986). Version modeling concepts for computer-aided design databases, *Proc. ACM-SIGMOD Conf.*, Washington, D.C., 379-386.

[KDM88] Kotz, A.M., Dittrich, K.R., Mülle, J.A. (1988). Supporting semantic rules by a generalized event/ trigger mechanism. *Proc. EDBT '88*, Venice, Italy, 76-91.

[KMS*89] Koubarakis, M., Mylopoulos, J., Stanley, M., Borgida, A., Jarke, M. (1989). Telos: a knowledge representation language for software information. Report KRR-89-01, University of Toronto, Ont.

[KP81] Koenig, S., Paige, R. (1981). A transformational framework for the automatic control of derived data. *Proc. 7th Intl. Conf. Very Large Data Bases*, Cannes, France, 306-318.

[KRÜG89] Krüger, E. (1989). Integritätsoptimierung in deduktiven Objektbanken am Beispiel ConceptBase. Diploma thesis, Universität Passau, W. Germany.

[LLOY89] Lloyd, J.W. (1989). Meta-programming for knowledge-based systems. *Proc. 3rd GI-Kongreß Wissensbasierte System*, Munich, W. Germany.

[MBW80] Mylopoulos, J., Bernstein, P.A., Wong, H.K.T.(1980). A language for designing interactive data-intensive applications. *ACM Trans. on Database Systems 5*, 2, 185-207.

[NCL*87] Nixon, B., Chung, L., Lauzon, D., Borgida, A., Mylopoulos, J., Stanley, M. (1987). Implementation of a compiler for a semantic data model: experience with Taxis. *Proc. ACM-SIGMOD Conf.*, San Francisco, Ca, 118-131.

[OSTE87] Osterweil, L. (1987). Software processes are software too. *Proc. 9th Intl Conf. on Software Engineering*, Monterey, Ca, 2-13.

[RJ89] Rose, T., Jarke, M. (1989). A decision-based configuration process model. To appear in *Proc. 12th Intl. Conference Software Engineering*, Nice, France..

[SA85] Snodgrass, R, Ahn, I. (1985). A taxonomy of time in databases. *ACM- SIGMOD Conf.*, Austin, Tx, 236-246.

[SHP88] Stonebraker, M., Hanson, E., Potamianos, S. (1988). The Postgres rule manager. *IEEE Trans. Software Eng. SE-14*, 7, 897-907.

[SKW85] Smith, D.R., Kotik, G.B., Westfold, S.J. (1985). Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. Software Eng. SE-11*, 11, 1278-1295.

[SS86] Sheard, T., Stemple, D. (1986). Automatic verification of database transaction safety. COINS Report 86-30, University of Amherst, Mass.

[TK89] Topaloglou, T., Koubarakis, M. (1989). Implementation of Telos: problems and solutions. Report KRR-89-08, University of Toronto, Ont.

[ULLM89] Ullman, J.D. (1989). *Principles of Database and Knowledge-Base Systems*, Vol. II, Rockville, Md: Computer Science Press.

[YOKO89] Yokota, K. (1989). What is expected of an object-oriented data model? In Ritter, G.X. (ed.): *Information Processing ´89*, North-Holland, 799-800.