A Decision-Based Configuration Process Model

Thomas Rose, Matthias Jarke
University of Passau, P.O.Box 2540
D-8390 Passau, W. Germany

Abstract
Configuration management is linked to a data model of software processes in order to clarify the relationship between programming-in-the-large, in-the-small, and in-the-many. The data model also allows for the separation of conceptual from document-based version and configuration management, by focusing on careful modeling of design decisions as a unifying concept to describe versioning, configuration, and mapping tasks. The close relationship to in-the-small ideas suggests applicability in process reusability. Besides describing the model, we report on a prototype implementation on the knowledge base management system ConceptBase and experiences in a real-world case study.

1 Introduction
As a part of managing and maintaining large-scale software systems one has to deal with consistency-in-the-small, consistency-in-the-large and consistency-in-the-many. Typically, these dimensions of consistency should be ensured by software development (sub-)environments dedicated to programming-in-the-small, programming-in-the-large and programming-in-the-many. Basically, it can be said that these kinds of environments focus on content-oriented issues, object management, and people, respectively.

Programming-in-the-small is a well formalized domain with substantial knowledge about tasks, methods, and tools to support them automatically. One thoroughly researched approach in this area is the style of computer-aided, intuition-guided programming [5]; formal transformations are selected by the user but executed by the system. Many similar works have appeared in the realms of formal methods and AI techniques in software engineering [1]. On the other extreme, group cooperation required for programming-in-the-many is not yet much more than an increasing research trend with limited SE-specific support by formalized models.

In between, programming-in-the-large seems to be intended to bridge the big gap between small pieces of programs developed with formalized support and large accumulations of programs developed by teams with diverging requirements and goals. However, bridging this gap should not result in hacking an integrated environment with arbitrarily selected features. In our opinion, programming-in-the-large and in-the-many must be tackled in a software engineering-like manner, i.e. one has to determine the requirements, develop a conceptual design of the data types and operations, and code this design to gain an implementation as effective and efficient as possible.

Our hope that an integrated environment can be built in this style derives from the observation that consistency-in-the-small, consistency-in-the-large and consistency-in-the-many share a conceptual focus - objects or modules [25]. For programming-in-the-small, modules form the external definition of what content-oriented work is supposed to achieve. For programming-in-the-many, module definitions form the unit of negotiation and contracting; modules can also be exploited to define ownership relationships between software development subgroups and their products.

Programming-in-the-large [29] is geared at providing common object management facilities that allow in-the-small work to position itself appropriately within the history of a large development project, and that relieve in-the-many work from technical details. Of necessity, programming-in-the-large cannot concern itself with details of the content in an environment with possibly heterogeneous languages, methods, and tools. It must provide a data model that reflects and supports the global, language-independent design decisions, typically dealing with versioning and configuration issues. Thus, one could claim that version and configuration management is essentially a database task, albeit one which requires a lot of active decision support components not present in today's DBMSS [2, 32].

Our approach assumes a software process in which design decisions are organized according to some explicitly known pattern (called "process programming" in [23]) and supported by computerized tools. When documented appropriately, such a process generates a rich model of its history and of the concepts used resp. developed.

In the literature, the notions of configuration and version are somewhat indistinct. The application of the DAIDA software process data model [13] helps to clarify their different roles. Our model is designed to distinguish between the outcome of configuration and version activities, the activities building up configuration and version processes, and the tool assistance for these activities. Activities and the corresponding tool assistance are modelled in terms of an object-oriented notion of decisions.
We propose a concept-based specification of version and configuration management. The specification of conceptual configurations apply conceptual knowledge about the software objects and development methods, independently of the representation of software objects by documents. This allows a separation of concerns between the conceptual knowledge about software objects and their physical representation. For example, this paper describes a physical representation by partially reusing existing source document management tools as document configurations. The concept-based specifications as well as the process of implementing these concepts are modelled in terms of the software process data model (fig. 1-1).

![Diagram of Conceptual Configurations and their Implementations](diagram.png)

**Fig. 1-1:** Modeling conceptual configurations and their implementations

The application of this approach is not restricted to programs but allows one to specify and document configurations of arbitrary life-cycle objects such as requirements, designs, etc. The configuration of requirements, designs and implementations to consistent development histories is intended as a means for the reusability of software processes, thus forming a basis for a tool that may produce software by the configuration of requirements.

Section 2 contains a brief summary of approaches proposed in the literature for dealing with consistency-in-the-large. Our approach is outlined in section 3, and presented in detail in section 4, using a knowledge representation language for formalization and a process reusability example as an illustration. Section 5 presents a prototype assistant which allows the graphical animation and interactive specification of conceptual configurations and the execution of their implementations. Section 6 reports a real-world experience: the portation of a system of approximately 100 module groups from a UNIX-SUNView to a VAX/VMS environment. Finally, section 7 presents conclusions and suggests future work.

### 2 Previous Work

When database people think of consistency-in-the-large, they think of schemas, database states, integrity constraints, transactions and similar general concepts [3]. They frequently overlook that, while these concepts are essential, more semantics from the domain of software engineering is needed to provide real help [32]. It is not sufficient to know that versions and configurations exist but also how and why they came about.

Within the software engineering literature, two directions of work have tried to achieve a richer model of consistency-in-the-large, one closer to in-the-many requirements, the other closer to in-the-small scenarios.

In the first approach, it is claimed that teams require coordination which cannot be expressed adequately by integrity constraints on objects and operations alone. Group coordination is based on contracts about tasks to be performed, and negotiation processes about goals to be achieved (e.g., [18]). Thus, consistency is ensured by negotiation.

The second approach focuses on decisions and/or system description networks (e.g., [21, 24]). System status and evolution are represented in terms of system, task and activity descriptions. Deliberations may be documented, but these negotiation processes are not actively involved in the in-the-large view of software processes.

Both approaches require a software information system as a logically central repository to identify, control, audit and account a system's status and evolution even when it is distributed physically. Traditionally, such software information systems are referred to as version and configuration control systems [37]. Tools and techniques proposed for these systems can be characterized along three dimensions:

- data models to describe and store software systems,
- update behaviour to represent the reactions on modifications to software systems, and
- representational adequacy, i.e., semantic expressiveness.

The data modeling dimension concerns architectural descriptions of software systems. Early tools like SCCS [31] and the more recent RCS [34] and CMS [8], contribute to efficient version storage and history recording. Versions are associated with a fixed set of attributes (mainly version numbers) for identification. Structural descriptions were introduced by the MAKE [11] tool by means of dependency relations. Additionally, MAKE incorporates product manufacturing rules and efficient system re-building based on the date attribute of files. Compatibility constraints of configurations must be handcoded in terms of version numbers, and versions of configurations must be represented by versions of MAKE files.

These early systems can be characterized by a restricted, file-oriented data model and ad-hoc languages for specifying and achieving compatibility. Systems such as ADELE [2] and the software database CACTIS [13] pioneered the use of standard predicative constraints or attributed grammars, whereas standard data base models such as the relational model for handling structural descriptions were experimented with in systems such as SIO [4]. More recently, the use of object-oriented modeling schemes has been proposed which appear to have the potential to integrate several desirable features. There are also specialized data models for version / configuration managers employed, e.g., in the VLSI area [7] and in the area of SE databases (DAMOKLES [30], IRIS [19], and others).

**Update behaviour** modeling has to cover modifications of individual versions, configuration structures and compatibility constraints; essentially, one has to define not only when a new version of a configuration disturbs the "harmony" of the software system but also how to react to such problems.

A modification of an individual version requires a check whether the new version accomplishes the generic version description. For example, a formal notion of upward compatibility [22] can be introduced that preserves the consistency of the entire system in case of substitution; however, testing upward compatibility between two arbitrary objects is undecidable and thus unsuitable for automation. The Inscape environment uses program verification techniques to check limited semantic properties of versions [27].

In general, modifications may require the propagation of changes or at least the delivery of change notifications to human or technical actors. For instance, smart recompilation [35] detects and propagates necessary recomplimations; it is designed to minimize recompilation efforts in case of program modifications. More general, ADELE supports propagation and notification by means of triggers activated by a constraint violation.

Until now, we discussed system reactions to modifications of existing configurations. A more basic application of constraints and rules for update handling concerns the derivation of configurations. ADELE allows the local definition of constraints with individual version descriptions; the selection of
an individual version leads to a modification of the constraints to
be fulfilled by other versions. In particular, when configuring
systems in a hierarchical style, local definitions lead to a
constraint propagation along the entire system. On the other
hand, the configuration manager of SIO [20] facilitates the
definition of configuration rules for global contexts. Compati-
bility constraints can be expressed by relational queries and
deductive rules, applying the theory of deductive databases to
detect inconsistency. If there is no exact match for a query,
preferences are used to select objects [20].

These discussions already involve the issue of
representational adequacy. Early systems are based on
version numbers and procedural programs working on "units"
[26], i.e., uninterpreted source files. In more recent proposals,
attributes associated with file version descriptions can be
exploited for a more controlled version and configuration
management, using either purely syntactic comparison or even
some semantics of the attributes.

However, viewed from the approach advocated in this paper, these extensions mix two concerns: the management of
units of storage and the management of content-oriented
concepts.

Another question of representational adequacy concerns the
relationship of object management (version and configuration
control) to the in-the-small and in-the-many aspects. Like [4],
we see a close relationship between software process modeling
and configuration management; specifically, we wish to
distinguish outcomes, processes, and tools of configuration
management. A deeper integration with in-the-small issues can
be reached when we are able to configure not only the outcomes
of previous software processes (reusable source or object
codes) but also their underlying development histories (process
reusability).

3 Overview of our Approach

Based on an analysis of previous research as outlined in the last
section, we defined three requirements to improve version and
configuration management:

(a) Integration of software process modeling and configuration
management: Effective configuration management requires the
application of knowledge from the underlying software
process due to representational adequacy and semantic
expressiveness.

(b) A software knowledge representation language: Representing the outcomes and the decision history of
software systems requires a knowledge representation
language to describe software objects as well as their
evolution appropriately. This software knowledge must
serve as a basis for configuration control.

(c) Separation of concepts and documents: From a
methodological point of view, one needs to separate the
specification (concept) and the implementation (document)
of configuration and version tasks.

Our approach is based on a software process data model which
is able to represent software to be configured as well as
software to configure. We propose to partition the software
process knowledge base into more abstractly described modules
or objects, to allow the independent development of such
objects along documented version histories, and to construct
revisions or variants of complete systems by configuration of
module versions. In our case, this not uncommon approach [7]
is realized in the following steps:

(1) Choice of a suitable modularization concept: From a
database viewpoint, the traditional approach for
modularization in software engineering, namely to leave
the modularization decisions completely to the human
designer, has the disadvantage that it can be difficult to find
for each piece of new knowledge or combination of rules it should
belong to. We have therefore found very attractive the
"worlds" approach taken in [36] which proposes a
knowledge base partitioning schema like those in
distributed databases. On the other hand, such a model is
difficult to combine with existing modularization concepts
at the programming level. In our current implementation,
we have circumvented the issue by using a sufficiently
course level of granularity. Thus the in-the-small modu-
larization based on traditional module concepts and
in-the-large modularization based on database concepts do
not interfere; but there is clearly much more research
needed to achieve a deeper understanding of these
interactions. Regardless of the specific choice of module
concept must satisfy the following safety condition:
Consistency = Cons-in-the-large + Cons-in-the-module +
Cons-in-implementation.

(2) Modeling versioning and configuring activities: In our
approach, version and configuration management is
viewed as a software process. Its activities are modeled as
executing decisions: A specific conceptual version and
configuration model consists of generalizations of
theories of such decisions which allow the inheritance of certain
properties from more general to more specific decision
classes. For example, such higher-level classes may define a
common physical storage or configuration context
of such decisions which allow the inheritance of configuration
consistency. The implementation of these
constructs also supports triggering mechanisms that serve
as a basis for powerful conflict handling mechanisms.

(3) Developing a formal basis for a configuration assistant:
This assistant supports the correct instantiation of
configuration decision classes -- the construction and
efficient reconstruction of actual configurations. The
knowledge representation language at the conceptual level
offers deduction rules for version selection and configu-
ration decisions, and integrity constraints for checking
configuration consistency. The implementation of these
constructs also supports triggering mechanisms that serve
as a basis for powerful conflict handling mechanisms.

(4) Achieving multi-level consistency: The above-described
part of the assistant only secures consistency at the
conceptual level. But, as mentioned earlier, one distin-
guishing feature of our approach is that we have separated
the conceptual from the physical representation. This is
another case of applying the software process data model.
Relationships between the levels are modeled as mapping
decisions; specific transformational tools for automating
the mundane parts of this mapping are built.

(5) Arriving at an architecture of the configuration assistant:
The above hierarchy implies several subhierarchies. Existing
version and configuration tools to be reused for storage
management must be modeled in the same language as the
conceptual model. The configuration assistant can thus
concentrate on the conceptual-to-storage mapping within a
given language since translation issues are factored out.
Consistency must be tested at each level, so that we again
have the kind of overall consistency formula under (1):
consistency-in-the-large = conceptual-cons +
mapping-cons + physical-cons.

Until further study, we believe that consistency proofs at
the higher levels can only be used to partially optimize
away the proof needs at the lower levels but that some
remainder will stay in practice.
The final goal is an "intelligent" configuration assistant in which the user graphically sketches a (possibly inconsistent and incomplete) conceptual configuration and the assistant helps in making it consistent and mapping it to physical tools. We have gained practical experiences with several substeps of this overall plan and developed a knowledge-based formalism for dealing with the remaining parts. Both will be described in the remainder of this paper.

4 A Configuration Process Data Model Based on Versioned Objects

In this section, we formalize and illustrate the decision-based approach to configuration process management outlined above. We first review the main properties of a software process data model developed in the ESPRIT project DAIDA, and then show how it is applied and extended in designing and implementing conceptual version and configuration management.

4.1 Preliminaries

The software process data model proposed in [15] concentrates on data modeling aspects of software environments. In its kernel, it is not intended to cover all the aspects of in-the-small, in-the-large, and in-the-many work but relies on the extensibility of the underlying knowledge representation language to specialize it to these tasks as needed. An application to in-the-small is reported in [16], whereas [19, 33] describes early work on in-the-many extensions. Both the kernel model and its extensions are implemented on top of the ConceptBase [10] implementation of the knowledge representation language CML/Telos [17].

Disregarding the above extensions, there are three dimensions of the model to consider (fig. 4-1). The first dimension (the process data model proper) models a software process as a history of tool-aided executions of decisions that transform input objects into output objects; this model is similar to the structure-policy-mechanism metamodel proposed in [28] but differs from most other models (e.g., those based on the entity-relationship approach [24]) by its explicit focus on tool usage. Using the powerful attribution mechanism of Telos, complex semantic descriptions can be attached to objects, decisions, and tools. Of specific interest are dependencies which relate attributes of input objects to attributes of output objects, thus facilitating incremental propagation of change by a data structure similar to those of truth maintenance systems [9].

The second dimension (conceptual level) is also facilitated by a feature of Telos, namely its infinite hierarchy of instantiation relationships; that is, a Telos knowledge base can be organized in an infinite hierarchy of tokens, simple classes, metaclasses, metametaclasses, etc. This feature proves to be an invaluable basis for extensibility. In fig. 4-1, the highest level (base model) describes the general concepts of decision, object, and tool. It can be instantiated by the definition of a particular software environment (in which this paper is a particular conceptual configuration process model), and thus serves as a simple kind of software process model "generator". In turn, the model of a given environment can be instantiated with the description of a particular software process. If needed, this project model can be again instantiated by example data, e.g., for testing some intermediate version of the system.

Finally, the third dimension (life-cycle) concerns the central role of multiple representational levels of all the above components. Our experiences as well as conventional wisdom in software engineering have clearly shown the need for distinguishing the requirements, the design, and the implementation of software process concepts, and for representing the mappings among these [14].

We believe that this separation can even usefully transcend to the in-the-many extensions. For example, the requirements level of programming-in-the-many is modeled by contracts, the design level by Petri net-like plans or scripts, and the implementation level by human-human and human-computer interactions documented by messages.

4.2 Versioning and Configuring as Tool-Aided Decision-Making

Version and configuration management "is software too" [23]. The software process data model assists in representing the outcome, the activities and the tool-assistance of version and configuration tasks, i.e., it allows to model versioning and configuring as a process of tool-aided decision making.

Basic concepts to model configurations are: knowledge about the configuration as a whole (typically called interface), a set of component objects, a relationship between them and tools processing this relationship. Not only component objects of a configuration, but also entire configurations are represented as design objects. A design object representing an entire configuration views it as a monolithic (unpartitioned) object and merely represents the fact that there is an object. From the viewpoint of the application domain, one knows that this object needs to be configured in order to obtain its properties. Methods to configure objects into compound objects are represented by decisions. The from-objects of a configuration decision constitute the component objects. The to-object refers to the interface object. Thus, a configuration decision represents the relationship between an interface and the conceptual structure of its implementing object aggregation.

Design objects may come with semantic descriptions; therefore, compatibility assertions can be expressed in terms of semantic properties. Such properties are derived from the application domain but not necessarily intended to deal with program verification in-the-small.

Compatibility assertions are associated with decisions and not with configuration interfaces. Thus, we can distinguish between inconsistency in the system definition itself (it does not belong to any known class of software) and inconsistency of the configuration. Compatibility assertions are represented by Telos rules. Inconsistency is documented by making the attempted configuration member of a specialized exception class. This approach allows one to document inconsistent configuration decisions for future negotiation, completion or correction (similar to ADELE [2]). Successful instantiation of compatibility rules creates dependency objects between component and interface objects. These dependencies might be used to create change notification or propagation.
programming language process data model. The project represented by corresponding classes from left to right. Fig. 4-2 shows an example of our approach, with the basic process model depicted at the top. The middle layer gives a specification of an example configuration decision class (i.e. a development method) -- the three levels of the DAIDA representation: requirements modeling in a knowledge representation language (World), conceptual design in a semantic modeling language (Design), and implementation in a database programming language (Impl). An entire development status configured by the decision Configure_IS_Dev is represented by IS_Dev. The configuration of a particular information system project is shown at the bottom -- the project level of the software process data model. The project represented by EP_Manager is to configure an information system for the assignment of Employers to Projects. In the remainder of this section, we shall look at this example in more detail to illustrate the use of our model at the instance level.

Fig. 4-3 shows an example evolution of this information system at the in-the-small level. The DAIDA levels are displayed from top to bottom; the version history is shown in the columns from left to right. Fig. 4-4 gives a graphical representation of the conceptual version and configuration history for the same example. Each rectangle denotes a design object which presents a requirement, a conceptual design or an implementation object of fig. 4-3. These design objects are instances of the corresponding classes EP_REQ, EP_DES, EP_REL and EP_CONST in fig. 4-2, i.e. they are the prototypes for the project at the bottom of fig. 4-2 (cf. fig. 4-1). Ellipses refer to mapping and refinement decisions performed during the development of fig. 4-3. In the sequel, the verbal outline of the example concerns the development scenario depicted in fig. 4-3. Corresponding design objects and decisions which are presented in fig. 4-4 are written in type writer font (in parentheses).

In the initial requirements analysis (EmpPerPro_REQ) we consider a world with persons some of which are employees of a company. It is assumed that each employee works on at most one project; this is indicated in the requirements model by making the workson attribute single-valued. This model is mapped by a mapping decision (InitialReqMap) to a conceptual design (EmpPerPro_Des), which is mapped to a database schema (Impl_1), which consists of a relation (Rel_1) and integrity constraint (Con_1). In validating and testing this system concept, two major criticisms come up, resulting in decisions 1 and 2 of fig. 4-3.

First, one decides on a version of the requirements mapping leading to a new conceptual design (EmpPerPro_Des), which collapses the isA-hierarchy of persons and employees, i.e. each person is assumed to be an employee. This conceptual design is mapped to an implementation (Impl_2). In the second decision, it is noticed that employees may in fact work on more than one project, i.e. the single-valued constraint is removed in the requirements model. This decision leads to a new requirements model (MoreProj_REQ). Mapping to an implementation (Impl_3) via a conceptual design (MoreProj_Des) proceeds straightforward.

The following example configurations show how compatibility assertions can be improved when they can rely on additional knowledge about the development history, i.e. the software process model is integrated with configuration management (see also [4]). Thus, version selection is not only founded on (conceptual) static properties but also considers decision-based process dependencies. For instance, the decision to configure

\* EmpPerPro_REQ, EmpPerPro_Des and Impl_1 is consistent, since they are directly evolved from presumably correct mappings (initial setting in fig. 4-3);

\* but EmpPerPro_REQ, EmpPerPro_Des and Impl_1 is inconsistent, since EmpPerPro_Des relies on a revision of the requirements mapping of EmpPerPro_REQ and is therefore a variant of EmpPerPro_Des which does not share the design mapping with EmpPerPro_Des (setting after collapsing the isA hierarchy in fig. 4-3).

After some time, the company decides to retract its decision to allow employees to work on more than one project. Now one gets a requirement (revisedEmpPerPro_REQ) resembling the initial one (EmpPerPro_REQ). Therefore, the same mapping might be done leading to a design (SingleProj_Des). Syntactically, both requirements and designs are equal. But, conceptually they are different. SingleProj_Des is a conceptual successor of MoreProj_Des. The implementation mapping of EmpPerPro_Des cannot be reused because, otherwise, databases created since the third setting would have to be reorganized. Thus, the mapping of SingleProj_Des has to reuse the relation schema of Impl_3, together with an additional integrity constraint that enforces working on one project only. A system which considers the underlying development process can therefore derive that the configuration of

\* EmpPerPro_REQ = RevisedEmpPerPro_REQ, EmpPerPro_Des = SingleProj_Des and Impl_4 (reusing the relation of Impl_3) is consistent;

\* but EmpPerPro_REQ = RevisedEmpPerPro_REQ, EmpPerPro_Des = SingleProj_Des and Impl_3 is inconsistent.

320
4.3 Implementation of Conceptual Configurations

Conceptual configurations and versions refer to a concept-based specification of configuration and version management represented in our case by design decisions and design objects. Compatibility assertions apply the semantic descriptions of design objects, and of mapping or refinement decisions among design objects. This approach is not restricted to program sources but applicable to arbitrary software objects.

Implementable configurations and versions are provided by file or document-oriented tools. Fig. 4-5 contrasts conceptual and implementable decision classes for the case of versions. The conceptual model makes the difference between variants or revisions precise by documenting the intention of developers. Objects interrelated by refinement decisions (more projects or again only one in fig. 4-4) are revisions: versioning of design objects leads to revisions. The outcomes of two versions of a mapping decision (Collapse-isA), are considered variants: versioning of decisions leads to variants.
Fig. 4-5: Decision about versions of sources versus conceptual version decisions

Similarly, fig. 4-6 shows a decision hierarchy for configurations. Configuration decisions can be nested, for example, putting a horizontal configuration of requirements analyses on top of a vertical configuration of how requirements can be consistently configured with designs and implementations, leads to a concept for "process reusability", i.e., the reuse of mapping experiences gained by earlier development processes rather than just the reuse of the outcomes.

These hierarchies provide a basic set of classes to represent versions, configurations and versions of configurations at the level of concepts. When implementing conceptual versions and configurations, one could follow the classical development process, i.e. develop a system specification, derive a conceptual design and then code the design. Instead, we propose a methodology to reuse existing tools for implementing conceptual configurations. The motivation is twofold. First, a configuration management system must support heterogeneous environments to be able to serve as a logically central repository, and it must be open to integrate existing tools since existing data must remain available. Secondly, reuse reduces the implementation effort.

However, it is difficult to map the conceptual model to existing tools directly. Therefore, data structures and operations of source management tools are described in terms of the software process model. This decomposes the mapping problem into the task of concept mapping and a simple translation algorithm. From the viewpoint of a structured methodology, the conceptual model is viewed as the specification of an information system for version and configuration support. The implementation model is a language to code such systems. Thus, implementing conceptual configurations is a software process (cf. fig. 1-1).

The implementation process centers around a kernel concept hierarchy of versions and configurations like the one given in fig. 4-5 and 4-6. All version and configuration decisions are specializations of these objects. Therefore, this concept hierarchy is a subset which spans the entire conceptual model. It is called the ideal model. All conceptual decisions are mapped to it as far as their storage and user interface needs are concerned. For instance, a decomposition of a software object due to quality improvement, functional extension or portation needs will be represented by a certain decision at the conceptual level. At the level of implementation one may create new files to store versions of decomposed objects, create new groups or modify a group, modify dependency relations as for the MAKE file, etc. For each class of the ideal model, there are several ways to represent a conceptual decomposition at the implementation level. The way chosen depends on applicable implementations and the achievement of certain design goals.

To summarize, the ideal model provides basic decision classes which

- are required for the conceptual representation,
- are supported at the implementation level by existing tools,
- describe reasonable mappings from conceptual to implementation decisions.

Each environment-dependent class of conceptual version and configuration decisions may further refine the inherited implementation to cover dedicated design goals. Due to the inheritance mechanism in our object-oriented representational framework, this refinement is inherited by another even more specialized decision.

Fig. 4-6: Decision about source configurations versus decisions about conceptual configurations
5 A Prototype Configuration Assistant

A prototype configuration assistant has been implemented as an extension to the design-oriented knowledge base management system, ConceptBase [10]. ConceptBase consists of an object processor and an idea processor connected in a client-server architecture. The object processor provides the representational facilities of Telos [17] and controls the consistency of the knowledge base. The idea processor offers a hypertext-like interaction with the knowledge base. It allows graphical and textual browsing, and interactive editing of the knowledge base.

![Diagram of ConceptBase](image)

**Fig. 5-1:** ConceptBase ...

The software process model is part of ConceptBase's knowledge base (fig. 5-1). When interacting with this specialized knowledge base, one needs dedicated support. Especially, the evaluation and execution of decisions requires assistance. A decision assistant animates the graphical representation and interactive execution of decisions (fig. 5-2). Its decision processor supports the instantiation of decisions, checks their consistency with respect to compatibility rules, and supports the completion of incomplete proposals by rule application. The decision animator visualizes decisions and process dependencies.

![Diagram of Decision Assistant](image)

**Fig. 5-2:** ConceptBase with decision assistance

The specific extension to version and configuration assistance focuses on conceptual configurations and their implementation, and developing dedicated graphical animation facilities to assist the execution of configurations. The configuration assistant applies the decision processor and enhances the decision animator. The conceptual model and its implementation are represented as knowledge base objects applying the knowledge representation facilities of Telos (figure 5-3). Since Telos is an extensible language, the system can be further extended by new source management tools, implementation decisions due to changing design goals and types of conceptual configurations. Different kinds of configurations such as pure program configurations or vertical configurations of implementation hierarchies are supplied with distinct graphical animations.

![Diagram of Configuration and Version Base](image)

**Fig. 5-3:** Integrating knowledge about versions and configurations in ConceptBase

The decision assistant prototype allows several ways to present this information to the user. Fig. 5-4 shows a screen dump for the example in section 4 (cf. fig. 4-4). We have built it in the style of an argument editor to show the contributions and viewpoints (rationales for design decisions) of different users. Fig. 5-4 presents the contributions of an end-user and a designer to collapsing the isA-hierarchy (cf. fig. 4-4). The user opposes the initial requirements mapping and poses a revised one. Therefore, the designer poses to collapse the isA-hierarchy; but, he opposes this variant due to the perceived implementation efforts. The style of this tool is similar to gIBIS [6], but has the additional advantage that it relates to formal objects. The formal conversation theory behind this tool is given in [12].

![Screen Dump of ConceptBase Argument Editor](image)

**Fig. 5-4:** ConceptBase argument editor
6 Case Study: The Creation of ConceptBase

The initial version of ConceptBase was implemented in BIM-Prolog and SUNView on SUN workstations. ConceptBase was a variant running under VMS on a microVAX. The portation of the system seemed to be a suitable case study for our approach due to the following characteristics:

- ConceptBase consists of about 130 modules with dozens of versions each (the size of an executable configuration is approximately 2 MB of Prolog code).
- It exists in several conceptual versions (with or without integrated time calculus, integrity constraint compiler, source management facilities, or argumentation facilities).
- Both versions evolve dynamically and consistently through activities by a team of about 15 persons.

A conceptual model of ConceptBase and ConceptBase was built to gain experience and determine specific requirements for a configuration assistant. The conceptual model allows the derivation of different variants with respect to specified requirements. For instance, several versions can be configured with respect to animation (e.g., icon definition and/or incremental layout for graphical browsing) and representational facilities (like deductive rules, time calculus or argumentation support) at the conceptual level.

In the case study, the main implementation work has focused on coupling the prototype carefully with CMS [8]. Fig. 6-1 shows a screendump, taken from the case study.

Two browsers display the implementation model of CMS (graphical browser to the left) and the documentation of all source modules of a large-scale software system within this implementation model (textual browser on the right side). Although this is not a matter of scientific progress, the browser helped us to navigate in our implementation. The development of a multi-criteria decision support facility to help the user express preferences for version and implementation selections within the ideal model consistently has also been finished.

When porting ConceptBase we noticed that current commercial source management tools lack concepts to represent the reorganization of a system. For example, in our VAX development environment, the window-based idea processor had to be replaced by an interface running on ASCII terminals.

To realize this with a minimum of reprogramming, several ConceptBase source modules had to be decomposed. In systems such as SCCS or CMS, these structural changes can be represented only by creating new source elements; justifications for these modifications cannot be represented at all. However, they are quite easy to represent at the conceptual level; even if they are then mapped to the same source structures as before, the conceptual process knowledge can be used for more adequate consistency checking and constraint propagation. Several similar examples seem to indicate the usefulness of our approach; we have also observed that it became much easier for new team members to become familiar with the system since the version and configuration model was introduced.
7 Conclusion and Future Work

We see three contributions in this paper. Firstly, we explicitly separated concept-oriented from document-based version and configuration management. The conceptual layer provides object and process descriptions connected to knowledge of the application domain. The implementation layer concerns the physical representation and document management. This separation improves the adequacy of representing configurations.

The second contribution is an object-oriented model to represent versions and configurations as specialization hierarchies of design decisions. Existing version and configuration decisions can be adapted to dedicated requirements. An ideal model provides basic conceptual decisions and their implementation by document-based decisions. Among other things, its specializations can be used to relate the in-the-large task to the specific in-the-small methodology at hand by appropriate compatibility constraints.

Third, the application of a decision-oriented software process data model enables a uniform modeling of the different tasks created by our approach: concept-oriented modeling, implementation-oriented modeling, and mapping from conceptual to implementation models. Thus, application of the software process data model allows an integration of domain knowledge from different application areas. Additionally, this uniform, decision-oriented representation enables the development of a common decision assistant to support the execution of version, configuration and mapping decisions.

Current work concerns the development of a dedicated configuration assistant, the refinement of the ideal model, and the specialization of the ConceptBase assertion language to version and configuration management tasks. These extensions require further practical studies to validate their usefulness.

Acknowledgment. The authors gratefully acknowledge the contributions of Manfred Jeusfeld to the ConceptBase implementation and the software process data model, of Michael Gocek and Hans W. Nissen to the prototype implementation of the approach proposed here, and of Carlos Maizlein for the integration with an argument editor for conversation support.

References