Towards a Logic-Based Reconstruction of Software Configuration Management

Matthias Jarke, Manfred Jeusfeld
Lehrstuhl Informatik V, RWTH Aachen
Ahornstr. 55, 5100 Aachen, Germany
{jarke,jeusfeld}@informatik.rwth-aachen.de

Michael Gocek, Andreas Miethsam
Universität Passau, Postfach 2540, 8390 Passau, Germany

The efficient configuration and re-configuration of software systems has been a major problem in software engineering. A large number of solutions has been proposed with different features and restrictions. In this paper, configuration management is presented from a logic perspective. It is shown that much functionality can be generated automatically when configurations are considered as views on a deductive database and recent results on deductive integrity checking, query optimization, and intensional updates are applied.

1 The Problem

Version and configuration management has been a major focus of research in practically all areas of computer-aided design, especially those like software engineering in which the whole design and production process happens within the system. Most research has been invested in the efficient *production and reproduction* of software components, e.g., in order to avoid excessive re-compilation efforts when only a small portion of a large system has changed. Complementary to this aspect is the *consistent (re-)construction* of a configuration, i.e., a construction process that satisfies a large set of different kinds of integrity constraints. Again, there is a strong implied efficiency goal here: you do not want to repeat all consistency proofs just because a small change on some component or (sub-)system specification has been made. Additionally, proof that certain properties have not changed may (but need not always) obviate configuration production work.

Specifically, according to [TICH88], the construction part of configuration management has to provide the following services:

- modeling of software architecture and component descriptions
- persistent and consistent management of version histories, recursively organized for base components, subsystems, and finally the overall system

This work was supported in part by ESPRIT basic Research Action 3012 (COMPULOG) and by the Deutsche Forschungsgemeinschaft in its program "Object Banks for Experts" (Ja445/1-3). A version of this paper appeared in Proc. 7th Knowledge-Based Software Engeneering Conf. (KBSE´92), McLean, VA, Sept. 20-23, 1992,pp. 132-141.

- version selection, component compatibility, and configuration compatibility for new configurations
- preservation of system consistency when any part of the system changes
- control of concurrency and sharing in the process of developing system ideas, distributing tasks, and allocating work results.

A large body of work exists in the area, mostly focusing on small subsets of the above problems. As a consequence, a deep understanding of the whole area appears still to be missing. Attempts to unify the different ideas and approaches are either just collections of required features and services structured according to some similar field such as compiler construction [TICH88], or they have come from graph theory [HK88b], using notions of attributed graphs to handle redundant derived data consistently [HK88a].

One conclusion of these works has been an emphasis on the *software process* [WILL88]. Any successful system will have to contain an explicit or implicit formal description of the versioning and configuring activities, not just the outcomes of these activities. In [RJG*91], it has been shown that the concept of a *design decision* is a good starting point for integrated models and process-oriented configuration process environments.

However, the above models still leave open the question how the decision processes are *formally* controlled. In a certain sense, this resembles the situation in databases a decade ago. As computational logic has made a significant contribution to enhancing our understanding of what a database is [REIT84], it is our research goal to achieve a similar result for software configuration management. This paper is a first attempt to summarize our results and experiments to date. It reviews a formal approach that uses recent results on deductive integrity checking and abduction-based intensional updates to formalize a software information base for configuration management and the operations on it; it also describes an experimental prototype implementation which has provided an interactive and incremental logic-based configuration assistant in order to gain deeper insights into the needs for meta-level modeling, optimization techniques, and services of the system [GOCE90].

A few researchers have observed the correspondence between specific configuration management tasks and certain services offered by deductive databases. However, each of these works only considers a small portion of the tasks:

- [MEYE85] proposes a graph-structured software knowledge base with a specific pre-defined set of link types that define the relationships between subsystems, components, specifications, etc. Similar proposals have also been realized in the software database area, e.g., in the DAMOKLES system [DGL86] or currently in the ITHACA Software Information Base [CJMV91].

- [AI87] show how the functionality of the Unix MAKE facility [FELD79] (i.e., to determine what configurations must be reproduced when particular components change) can be simulated elegantly by a Prolog program. But they do not consider the presence of integrity constraints beyond simple date attributes as in MAKE, nor the initial construction of consistent configurations beyond simply writing them down as deduction rules.
- SIO [BLLV87] uses deduction rules (with preferences) for component version selection but neither look at the reconfiguration task nor in any detail at the role that additional integrity constraints may play in the process.
- ADELE [BE87] uses different kinds of predicative integrity constraints to recognize consistency problems. But the user has to attach these constraints to the right objects (otherwise the system will behave erroneously), and the resulting corrective actions are hand-programmed.
- MARVEL [KBFS88] uses rules as a representation for external tool integration, with "opportunistic" rule invocation in either forward or backward fashion. There are no specific features for the configuration management problem.

Among other reasons, the moderate scope of these approaches may also be traced to the lack of efficient techniques for deductive integrity checking and intensional updates which would make a logic-based formalism for all of configuration management practically more appealing. New techniques such as proposed in [BDM88, BRY90, DECK89, KM90] enable a more comprehensive solution; in particular, the role of abduction is far better understood now than just two years ago.

From another angle, the concrete example of software configuration management may also contribute to the solution of some still persisting problems in using abduction for deductive database updates. We see two such problems: reducing the number of options in choosing abducibles, and strategies for the integration of integrity checking in the abduction process.

Addressing the former problem, early abduction approaches just draw updates from derivations by backward-chaining through deduction rules. Papers such as [DECK89, BRY90, KM90] propose techniques which restrict abducibles to those satisfying a set of integrity constraints defined over the deductive database. An additional idea typical for configuration management is to *reuse* as much as possible former abduction processes, thus further guiding the search for good solutions. In essence, the idea is to use existing configurations as second-class integrity constraints which should be violated "as little as possible", without having the same strength as "official" integrity constraints. This idea has also been expressed in slightly different contexts by [JR88, KAKA91], and is a cornerstone of the DAIDA methodology for development and maintenance of database application software [JMSV91].

The second question is how to integrate this restrictive information (integrity constraints and version history knowledge) in the search process. One possibility is to separate the concerns of abduction and deductive integrity checking: produce abducibles first, then use standard deductive integrity checking techniques to see whether they are acceptable -- a "generate-and-test strategy". A second possibility provides a deeper integration by compiling selected simplified integrity constraints into the deduction rules used for abduction -- a solution similar to "semantic query optimization". This second approach implies the possibility to provide specifications of new software components to be developed, even if the system cannot automatically generate the implementation of these specifications.

The case of software configuration management indicates that a combination of such strategies dedicated to the particular task at hand may be needed in practical situations; for example, most existing configuration systems distinguish between the roles of integrity constraints as "configuration constraints" (used for integrity checking of a particular kind of configuration), "completion rules" (used for abduction without much other integrity checking), and "compatibility constraints" (among components, may be dependent or independent of the particular configuration at hand). Metalevel control strategies may turn out to be useful to automate the assignment of general constraints to such classes, and to generate the corresponding dedicated evaluation procedures.

In the remainder of this paper, we present our logic-based approach to representation of and reasoning about configurations and describe a prototype implementation which supports some of these features.

2 The Approach

The aggregation of components can be formalized as special kinds of deduction rules called *configuration threads*. Compared to a "naive" approach, our rules contain explicit representations of the configuration decisions so that reuse can be reasonably well described. Software configuration management is then characterized by two search problems:

1. Re-use of components:
   Given the interface of a new system, find an implementation, if possible by re-using existing subsystems or components. Otherwise, provide a specification of what new components are needed, or suggest new configurations of existing components.
2. Re-configuration:
   Given a change on a component propagate this change to the affected modules, if possible without changing the interfaces of the systems that use this component.

The above problems are then solved by a combination of abductive and deductive integrity checking methods. The approach is illustrated with a simplified example from the configuration of DBMS software; the example is intensionally not presented at the code level but at the level of application-oriented requirements models. The idea is that actual code would be attached to these specifications by appropriate dependencies.

2.1 Representation of System Configurations in Logic

A system is recursively built up from module configurations. Each level is described by its interface - a set of properties and possibly constraints - and its components. The components can either be atomic objects or themselves be configured from smaller subcomponents.

Logically speaking, the interface specification of a system can be modeled as a consequence of its component specifications and of the decision how they were configured. The schema of such a so-called *configuration thread* is:

$$
\begin{array}{l}
\mathbf{interface}(if, p_1, .., p_n) \;<== \\
\quad \mathbf{configure}(co, if, c_1, ..., c_m) \text{ and} \\
\quad \mathbf{component}_1(c_1, a1_1, ..., a1_{j1}) \text{ and} \\
\quad ... \\
\quad \mathbf{component}_m(c_m, a1_m, ..., a1_{jm}).
\end{array}
$$

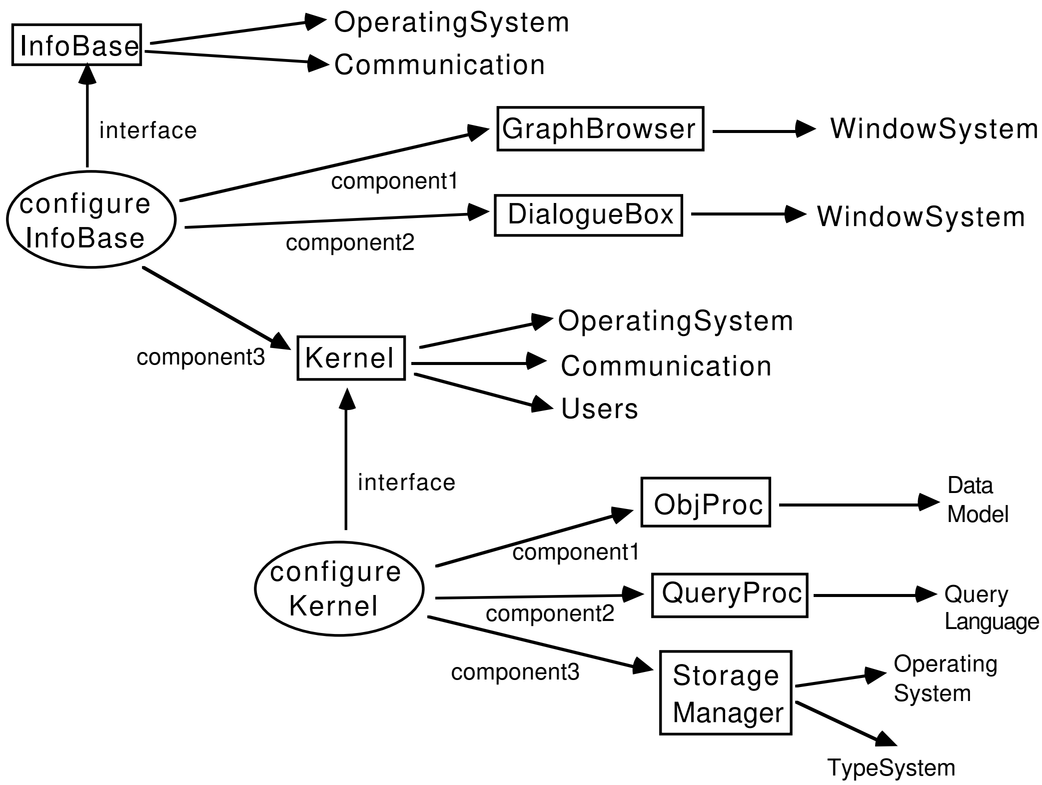
The first argument of the interface and component literals denote the identifiers of (software) items. The `configure` predicate represents the fact that the component modules c_1, ..., c_m and the interface module named `if` are aggregated to the complex object `co`, i.e. this predicate denotes the design decision to produce the software object. AND parallelism in the rule body ensures that all components exist. Multiple configuration threads with matching heads correspond to non-deterministic ways of implementing an interface (OR parallelism). The parameters p_i decribe the *exported* properties of module interface, whereas the properties ai_j are *imported* from their respective component modules. The components are either ground facts (already implemented) or there are configuration threads having them as conclusion.

For our user interface example, we assume that a version of the whole system `InfoBase` is characterized by a version number `c`, a specification of the operation system `os`, and a specification of the communication protocol `com`. The component `Kernel` has an additional slot for the maximum nuber of concurrent users. The next two components specify a graphial browser for window system `ws1`, and a dialogue box running under window system `ws2`.

```
InfoBase(c,os,com) <==
    configureInfoBase(cc,c,k,t1,t2) and
    Kernel(k,os1,com1,users) and
    GraphBrowser(t1,ws1) and
    DialogueBox(t2,ws2).

InfoBase(c,os,"debug") <==
    configureDebug(cc,c,k,t1) and
    Kernel(k,os1,com1,1) and
    DialogueBox(t1,"ascii").

Kernel(k,os,com,u) <==
    configureKernel(ck,op,qp,sm) and
    ObjectProcessor(op,dm) and
    QueryProcessor(qp,ql) and
    StorageManager(sm,os1,ts).
```



In the actual data model we use [JJ91], the interface and component description literals in the above formula are managed as views on a deductive object base which essentially consists of binary relations ("attribute objects", written `A(x,rel,y)`) organized according to an object-oriented structural schema. The above figure shows the `configureInfoBase` and `configureKernel` threads of the software configuration example. The rounded boxes represent design decisions that configure components to interfaces.

Configuration threads just describe the possible (at the class level) or actual (at the instance level) structure of a configuration. Essentially, they correspond to deduction rules used for a simple search process for reusable components without integrity checking, similar to the approach offered by MAKE. Since only structure is defined, the rules are pure Datalog with neither negation nor (normally) recursion. Consistency of configurations is described by integrity constraints over these structures. Most constraints make statements about the components and the interface attributes of a configuration. For simplicity of exposition of this paper, we assume that all variables are universally quantified, unless stated otherwise.

Definition 1

An integrity constraint matching the format

```
configure(co,if,c1,...,cm) ==> F
```

is called a *configuration constraint* for configuration `co`. `F` is a range-restricted formula. If the second variable `if` is missing in `F` then it is called a *compatibility constraint* for `co` (compatibility constraints could also be defined independently of particular configurations but this does not change much the problem or the method proposed below).

One can view such constraints to be restricted to the context of the configured complex object. The following constraints continue the example.

(IC1) The window systems of the graph browser and the dialogue box must be the same (compatibility constraint):

```
configureInfoBase(cc,_,_,t1,t2) ==>
(GraphBrowser(t1,ws1) and DialogueBox(t2,ws2) ==> ws1=ws2)
```

(IC2) The operating system of `InfoBase` should be the same as the one of its kernel (configuration constraint):

```
configureInfoBase(cc,c,k,_,_) ==>
(InfoBase(c,os1,_) and Kernel(k,os2,_,_) ==> os1=os2)
```

(IC3) The communication protocol of the InfoBase must conform to the communication protocol of its kernel and to the window system of its dialogue box (configuration constraint):

```
configureInfoBase(cc,c,k,_,t2) ==>
(InfoBase(c,_,com1) and Kernel(k,_,com2,_) and
DialogueBox(t2,ws2) ==> CompatibleComm(com1,com2,ws2))
```

The predicate `CompatibleComm` denotes a predefined table of comunication protocols and window systems.

2.2 Updates on the Interface Specification

A first problem of configuration management is the implementation of a new system. Such a configuration process should reuse existing components where possible, and the search for such components can be done by deductive query processing (possibly with similarity matching which we shall ignore here since they are besides the point we are trying to make). However, there are two problems with this process.

Firstly, the search should not be done separately for each component but existing sub-configurations should be reused as a whole. Moreover, if the search fails, deductive query processing would come up with an empty answer even if only one of dozens of components were missing. This is why we extend the search process by an abductive component: when the search for existing components fails, it generates a mock-up object which in effect is the formal specification of a missing piece which has to be found outside the software database or implemented from scratch. Moreover, when a successful configuration is built, this fact is recorded in the database. Summarizing, we get the following procedure (which we have implemented in Prolog):

> The insertion of an interface specification leads to a search on the past configuration decisions for this interface (see first condition literal of a configuration thread). If no matching interface exists then the search is passed to the component literals. If existing components are found then a new instance of the configure literal has to be inserted. Otherwise, abduction is applied for the insertion of new components.

Any inserted literal may be subject to integrity constraints. Since inserted literals correspond to the programming of new software pieces it is too costly to insert many new literals via abduction and then check whether they fulfill the requirements. Therefore, we propose to guard each insertion of a literal by its simplified integrity constraints.

Example 1: Let the component base consist of the following already implemented modules:

```
{InfoBase(IB-V2.0,Unix-V,IPC),GraphBrowser(GB-V1.0,X11-4),
 DialogueBox(DB-V1.0,X11-4),DialogueBox(DB-V2.0,X11-5),
 Kernel(K-V1.4,Unix-V,IPC,11),Kernel(K-V1.5,Unix-B,IPC+,12),
 configureInfoBase(CB1,IB-V2.0,K-V1.4,GB-V1.0,DV-V1.0)}
```

- The task is: **`Insert(InfoBase(c,Unix-V,IPC+))`** where (IC3) must hold
  --> configuration `CB1` does not apply since (IC3) is violated

- Resulting subtasks are:
  T1: **`Insert(Kernel(k1,os1,com1,u1))`** where `os1=Unix-V` (IC2) and `com1` fits to the communication protocol of InfoBase (IPC+) and the window system of the GraphBrowser (IC3),
  T2: **`Insert(GraphBrowser(t1,ws1))`** where `ws1` fits to the communication protocols of the kernel and of InfoBase (IC3),
  T3: **`Insert(DialogueBox(t2,ws2))`** where `ws2=ws1` (IC1),
  T4: **`Insert(configureInfoBase(cb,b,f1,w1,w2))`**

  Solve T1 (`Kernel`):
  --> let´s assume `CompatibleComm(IPC+,IPC,X11-5)` holds
  --> re-use `Kernel(K-V1.4,Unix-V,IPC,11)`

Solve T2 (`GraphBrowser`):
--> the existing solution `GB-V1.0` does not apply since
`CompatibleComm(IPC+,IPC,X11-4)` does not hold
--> a new graph browser has to be implemented that conforms the requirements; let `GraphBrowser(GB-V1.1,X11-5)` be the result

Solve T3 (`DialogueBox`):
--> the existing solution `DB-V2.0` applies --> re-use it

Solve T4 (`configureInfoBase`):
--> implement a new module version IB-V2.1 of the interface
`InfoBase(IB-V2.1,Unix-V,IPC+)` by importing the part modules
`K-V1.4`, `GB-V1.1`, and `DB-V2.0`
--> store the resulting configuration

```
configureInfoBase(CB2,IB-V2.1,K-V1.4,
                      GB-V1.1,DB-V2.0)
```

- Store the new interface `InfoBase(IB-V2.1,Unix-V,IPC+)`.

2.3 Updates on Component Specifications

In the previous section it was shown how an update to an interface is propagated downwards to the components. The same update may however have consequences to the complex objects that have this object as component: violations of configuration and compatibility constraints, and dangling interface objects that have no implementations any more. Both can be handled by deductive integrity checking.

Updates can be deletions, insertions or combinations of both; since component literals occur negatively in the configuration and compatibility constraints, they can only be violated during insertions (of new components or their attributes). Thus, we get the first part of the algorithm:

(1) An insertion on a component `c` violates a configuration or compatibility constraint.
The simplest reaction is to reject this update. A more clever solution is to find out whether the old software base is re-configurable:

--> Check by forward evaluation of those configuration threads with `c` as condition which interfaces are no longer configurable (technique is described in [JJ91]).
--> Try to create these violated interfaces by abduction.

For detecting dangling interface objects we build the logical completion (*only-if-halves*) [CLAR78] of the rules defining one configuration thread. This is justified since only such interface instantiations are considered for which at least one rule body is satisfiable. The only-if-half now serves as an additional integrity constraint for maintaining interfaces through versioning updates. They have to be evaluated on the case of component deletions since then an existing interface may no longer be valid. This gives the second part of the algorithm:

(2) A component `c` is deleted that occurs in a configuration thread.

--> View the only-if-half of the configuration thread as integrity constraint and check whether the simplified form for this update is fulfilled.

--> If yes: the update does not affect the interface.

--> If no: react as in (1).

Returning to the example, the only-if-halves of the `InfoBase` and `Kernel` configuration thread are:

(IC4) A `InfoBase` interface can exist only if the respective components given by its configuration thread exist:

```
InfoBase(c,os,com) ==>
   (∃ cc,k,t1,t2,os1,com1,u,ws1,ws2
     configureInfoBase(cc,c,k,t1,t2) and
     Kernel(k,os1,com1,u) and
     GraphBrowser(t1,ws1) and
     DialogueBox(t2,ws2) )
   or
   (∃ cc,k,t1,os1,com1,u
     com = "debug" and
     configureDebug(cc,c,k,t1) and
     Kernel(k1,os1,com1,1) and
     DialogueBox(t1,"ascii") ).
```

(IC5) A `Kernel` interface can exist only if the respective components given by its configuration thread exist:

```
Kernel(k,os,com,u) ==>
   (∃ ck,op,qp,sm,dm,ql,os1,ts
     configureKernel(ck,k,op,cr,bd) and
     ObjectProcessor(op,dm) and
     QueryProcessor(qp,ql) and
     StorageManager(sm,os1,ts) ).
```

Example 2: Let the component base from example 1 additionally contain:

```
{ObjectProcessor(OP-V1.0,nested),
ObjectProcessor(OP-V2.0,nested),
QueryProcessor(QP-V1,IPC),
StorageManager(QP-V1.7,Unix-V,IPC),
configureKernel(CK1,K-V1.4,OP-V1.0,QP-V1.7,SM-V1.3) }.
```

- The task is: **`Delete(ObjectProcessor(OP-V1.0,nested))`**
  --> configuration `CF1` does no longer hold since (IC5) is violated
  --> `Kernel(K-V1.4,Unix-V,IPC,11)` "dangles".

- We react on the integrity violation by starting reconfiguration of the dangling interface: **`Insert(Kernel(K-V1.4,Unix-V,IPC,11)).`**

- Resulting subtasks are:
  T1: **`Insert(ObjectProcessor(op,dm)),`**
  T2: **`Insert(QueryProcessor(qp,ql)),`**
  T3: **`Insert(StorageManager(sm,os,ts)),`**
  T4: **`Insert(configureKernel(ck,K-V1.4,op,qp,sm))`**
  Solve T1 (`ObjectProcessor`):
  --> take the second version `ObjectProcessor(OP-V2.0,nested)`

  Solve T2 (`QueryProcessor`):
  --> take the old solution `QueryProcessor(QP-V1.7,IPC)`

  Solve T3 (`StorageManager`):
  --> take the old solution `StorageManager(SM-V1.3,Unix-V,IPC)`

  Solve T4 (`configureKernel`):
  --> store the resulting configuration
  `configureKernel(CF2,K-V1.4, OP-V2.0,QP-V1.7,SM-V1.3)`

- Interface `Kernel(K-V1.4,Unix-V,IPC,11)` remains unchanged.

As a heuristic, the old configuration decision is replayed where possible (see tasks T2, T3, T4). The example shows that the propagation of the update can be kept local to the `Kernel` subsystem since the interface is left unchanged. A problem not addressed in this paper is how the costs of an update can be kept small, esp. how the number of component creations can be limited. A widely used heuristic in software engineering is to take a "similar" component and create a slightly different version that fulfils the requirements. This technique has the additional benefit that the integrity checking can be restricted to the "difference" between the old and new version provided all integrity constraints have the form of definition 1. Another technique is to use a base of generic components that can be parameterized by the required component attributes.

3 The System

To gain insight into the required structures and functions of a deductive database for configuration management, an interactive logic-based configuration assistant (hereafter: CMA) for the incremental construction and re-construction of consistent configuration descriptions has been implemented [GOCE90] as an extension of the deductive object base *ConceptBase* [JARK91]. It consists of a data model along the lines sketched in the previous section, and a graphics-based interface. We briefly describe the system and illustrate it with screendumps from an example session concerning the self-application of the assistant to its own configuration.

The CMA data model provides the recursive structure of modular object configurations discussed in section 2.1:

- interface specifications
- component specifications which, at the lowest level, may be descriptions of existing software components, otherwise, they are themselves interface specifications
- configuration decision objects that link interface and component specifications.

As usual in databases, this structure exists at both a schema level and an instance level (though both are treated uniformly as objects in amalgamated style). The instance level documents the actual configuration decisions that have been made for future reuse. The schema level defines possible configuration threads and the associated integrity constraints. The CA divides these constraints in three categories depending on the role they play in the interactive incremental configuration process:

- *Configuration constraints* restrict the possible composition of component versions; their role in the development process is initially passive but they can be used in semantic query optimization style upon user request.
- *Completion rules* actively select applicable component versions or point out places where abducibles (i.e., specifications of missing subsystems) have to be introduced.
- *Compatibility constraints* restrict what components fit together in any of a larger class of configurations. While the other two kinds of constraints create dependencies among (properties of) interface and components, compatibility constraints create dependencies between components, not necessarily involving a particular interface specification. Again, compatibility constraints can be used either passively to check a proposed configuration thread instance (generate-and-test), or actively to propagate constraints in selecting applicable versions of components to be included in such an instance (incremental semantic query optimization).

The model maintains these dependencies explicitly as object structures, and attaches the simplified forms of rules and constraints to be tested with particular user operations to these dependencies by triggers. In accordance with our general strategy for implementing integrity maintenance in deductive object bases [JJR89], the generation and maintenance of simplified forms is thus offloaded from update time to schema evolution time. The screendump above shows the dependency classes associated with the CMA configuration process as vertical arrows between attributes of the interface resp. component specifications. The practical meaning of such dependencies is that they function as triggers of the associated simplified forms of integrity constraints when updates on the attributes they point to (or start from) occur.

< SCREENDUMP 1>

Besides inheriting general ConceptBase tools such as the dependency browser shown above, the *CMA user interface* mainly consists of a dedicated configuration management window with four subwindows (see screendump below). The *main subwindow* (lower right) gives an status overview of the particular configuration task at hand. On its left, the class structure consisting of one particular configuration thread is shown (color: black). On the right, related to the class level by the dashed instantiation links, the available instances (actually, only a pre-selected subset of no interest to this discussion) of components and already produced configurations are shown. At any given moment, it is assumed that the user works out a new configuration instance; the status of this work is shown by highlighting the component instances in various shades to indicate: version selections proposed by completion rules or semantic query optimization; manual selections made by the user between available choices; and inconsistencies detected by checking configuration or compatibility constraints. Such inconsistencies are further explained in the error window (directly below the buttons). Moreover, the user can display any specification in a Display subwindow (lower left) to understand better the situation. He can also experiment with changes of either the interface or any component specification in the Editor subwindow (upper left), and trace their consequences to the consistency of some configuration thread instance in the main window.

<SCREENDUMP 2>

Different kinds of operations (which in turn trigger different kinds of simplified integrity constraints) are associated with the various object types in the main window as pop-up menu choices. Interface and component specification instances can be

- selected for participation in a configuration instance, with or without causing corresponding constraint propagation w.r.t. the remaining components
- excluded from selection for future constraint propagation applications
- replaced by earlier versions in the revision history to reconstruct old system versions
- explained by showing the full specification in the Display window
- changed using the Editor window.

A proposed configuration decision instance can be

- checked for configuration consistency / compatibility of already selected components
- completed tentatively with additional components according to completion rules
- effectively constructed as an existing configuration in the knowledge base
  (i.e., actual insertion of a proposed abduced object).

4 Conclusion

As stated in the introduction, several earlier systems have advocated the use of logic in configuration management. Our approach advances over these attempts by addressing the problem of missing components, the containment of change with subsystems, the separate representation of configuration structure and constraints, and the automatic generation of some of the triggers that have to be hand-coded in systems such as ADELE. This advance is made possible by adapting and combining ideas from abductive reasoning and deductive integrity checking with those of deductive query processing. Thus, an interactive and incremental approach to consistent configuration management can be supported by logic, a first step towards a logical reconstruction of software configuration management at a conceptual level.

References

[AI87] Asirelli, P., Inverardi, P. (1987). Enhancing configuration facilities in software development: a logic approach. *Proc. First European Software Eng. Conf.,* Strasbourg, France, 59-67.

[BDM88] Bry, F., Decker, H., Manthey, R. (1988). A uniform approach to integrity constraint satisfaction and satisfiability in databases. *Proc. Intl. Conf. Extending Database Technology,* Venice, Italy.

[BE87] Belkhatir, N., Estublier, J. (1987). Software management constraints and action triggering in the ADELE program database. *Proc. First European Software Eng. Conf.,* Strasbourg, France, 47-58.

[BLLV89] Bernard, Y., Lacroix, M., Lavency, P., Vanhoedenaghe, M. (1987). Configuration management in an open environment. *Proc. First European Software Eng. Conf.,* Strasbourg, France, 37-45.

[BRY90] Bry, F. (1990). Intensional updates: abduction via deduction. *Proc. 7th Intl. Conf. Logic Programming,* Jerusalem.

[CJMV91] Constantopoulos, P., Jarke, M., Mylopoulos, J., Vassiliou, Y. (1991). Software Information Base -- a server for reuse. Submitted to *VLDB Journal.*

[CLAR78] Clark, K.L. (1978). Negation as failure. In Gallaire, H., Minker, J. (eds.): *Logic and Data Bases.* New York: Plenum Press, 239-322.

[DECK89] Decker, H. (1989). Drawing updates from derivations. Report, ECRC, Munich, Germany.

[DGL86] Dittrich, K., Gotthard, W., Lockemann, P. (1986). DAMOKLES -- a database for software engineering environments. *Proc. IFIP Workshop on Advanced Programming Environments,* Trondheim, Norway.

[FELD79] Feldman, S.I. (1979). Make -- a program for maintaining computer programs. *Software: Practice and Experience 9,* 4.

[GOCE90] Gocek, M. (1990). Entwurf und Realisierung eines interaktiven Assistenten für Versions- und Konfigurationsentscheidungen. Diploma thesis, Univ. Passau.

[HK88a] Hudson, S.E., King, R. (1988). The CACTIS project: database support for software development environments. *IEEE Trans. Software Eng. 16,* 6, 708-719.

[HK88b] Heimbigner, D., Krane, S. (1988). A graph transform model for configuration management environments. *Proc. 10th Intl. Conf. Software Eng.,* Singapore, 216-225.

[JARK91] Jarke, M., ed. (1991). ConceptBase V3.0 user manual. Report MIP-9106, Universität Passau.

[JJ91] Jeusfeld, M., Jarke, M. (1991). From relational to object-oriented integrity simplification. *Proc. 2nd Intl. Conf. Deductive and Object-Oriented Data Bases,* München, 460-477.

[JJR89] Jarke, M., Jeusfeld, M., Rose, T. (1989). Software process modeling as a strategy for KBMS implementation. *Proc. 1st Intl. Conf. Deductive and Object-Oriented Data Bases,* Kyoto, Japan.

[JMSV91] Jarke, M., Mylopoulos, J., Schmidt, J.W., Vassiliou, Y. (1991). Information systems development as knowledge engineering: a review of the DAIDA project. *Programirovanie 17,* 1, 3-33 (Moscow, USSR: Academy of Sciences).

[JR88] Jarke, M., Rose, T. (1988). Managing information systems evolution through knowledge bases. *Proc. ACM-SIGMOD Conf.,* Chicago, Il.

[KAKA91] Kakas, A. (1991). On the evolution of deductive databases. Report, ESPRIT BRA 3012 (COMPULOG), Imperial College, London, UK.

[KBFS88] Kaiser, G.E., Bhargouti, N.S., Feiler, P., Schwanke, R.W. (1988). Database support for knowledge-based engineering environments. *IEEE Expert 4,* 2, 18-32.

[KM90] Kakas, A.C., Mancarella, P. (1990). Database updates through abduction. *Proc. 16th Intl. Conf. Very Large Data Bases,* Brisbane, Australia.

[MEYE85] Meyer, B. (1985). The software knowledge base. *Proc. 8th Intl. Conf. Software Eng.,* London, UK, 158-165.

[REIT84] Reiter, R. (1984). Towards a logical reconstruction of relational database theory. In Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (eds.): *On Conceptual Modelling*. New York: Springer-Verlag, 191-233.

[RJG*91] Rose, T., Jarke, M., Gocek, M., Maltzahn, C., Nissen, H.W. (1991). A decision-based configuration process environment. *Special Issue on Software Environments and Factories, Software Engineering Journal.*

[TICH88] Tichy, W. (1988). Tools for software configuration management. *Proc. Intl. Workshop Software Verson and Configuration Control,* Grassau, Germany, 1-20.

[WILL88] Williams, L.G. (1988). Software process modeling: a behavioral approach. *Proc. 10th Intl. Conf. Software Eng.*, Singapore, 174-186.