NelleN: a Framework for Literate Data Modelling

Michel Léonard
Ian Prince

Centre Universitaire d'Informatique
Université de Genève
1207 Genève
Switzerland

leonard@cui.unige.ch
prince@cui.unige.ch

Abstract

The literate data modelling paradigm provides a basis for structuring, justifying, and documenting the data modelling process. The paradigm is based on deliberation schemas consisting of issues, positions, and arguments. Deliberation schemas provide both a default argumentation space for decision-taking and a structure for recording the rationale once a decision is taken. An extendable class-based implementation framework, NelleN, provides a foundation for complementing a CASE-tool data dictionary with deliberation schemas. The framework consists of algorithm, deliberation schema and deliberation triggering classes. A subclassing mechanism is used for extending the framework. A small prototype demonstrates the type of CASE-tool that may be implemented using the framework.

1 Introduction

Traditional information system design methodologies such as DATAID-1 [4], Information Engineering [11], and Remora [19] have arisen from the need to manage the development of increasingly larger and more complex applications. These methodologies concentrate on the quality of the information system by promoting a strict sequence of refinement steps and formalisms and imposing validity rules on the specifications produced. Productivity of the development process is usually provided for by CASE tools (IEF [11], for example, supports the Information Engineering methodology).

A number of assumptions prevail behind most of the current information system design methodologies. One assumption is that systems are designed from scratch. Methodologies provide an analysis phase that prescribes an objective examination of the problem domain but most do not, however, provide methods for integrating the existing system with the new system to be developed. Instead, most methodologies concentrate only the creation of the new system. Methodologies also frequently assume that the requirements of the application domain are stable. This assumption fixes the problem statement at the beginning of the development process and excludes problem statement redefinition during the development process. Fixing the problem statement at the beginning of the development process assumes that the application domain's requirements can actually be described completely and correctly during the analysis phase. This presumes that complete and correct information is available during the analysis phase. The most important assumption — and the one that results from those previously mentioned — is the assumption that the development process is essentially linear. A linear development process, for example, prohibits the re-analysis of the application domain once the analysis phase has been completed. This assumption is reflected by methodologies frequently being based on the waterfall [14] model of development.

The consequence of the assumptions outlined above is that most methodologies and their supporting CASE-tool's stress *what* is produced by the method (the design products) and neglect *how* the design products are analyzed, refined and documented (the design process). This bias has lead to the following shortcomings:

- maintenance of the design product is rarely considered as an integral part of the design process because most methodologies only consider constructing a new system.
- design alternatives may not be easily explored because of the linear approach to the development process.
- undocumented assumptions must be made during the design process, severely hindering maintenance, because the methodology does not provide for the decision process.
- the design process is hindered by excessively pessimistic computer assistance because completeness and consistency of the design products is stressed.
- CASE-tools provide little support besides facilities for consistency checking, diagrams and documentation in the form of reports [2].

Design is essentially a complex iterative process that can not be determined *a priori*. Yet current information system methodologies and tools — because of the assumptions they are based upon — do not support this complex cognitive process. The most popular — and controversial — process model used for information system design is the waterfall model. The waterfall model, because it prescribes a linear process, has come under considerable criticism because in real-world design it is not practical — even possible — to anticipate all design issues during the early phases of the development process. Other models for the design process have been proposed to overcome the waterfall model's limitations: the spiral model of Boehm and models based on prototyping are just two. The focus, however, is still on supporting *what* is designed and not on *how* it is designed.

What are the elements of the design process and how may they be supported by computer-based tools?

Decision Making Any design process involves making decisions. Decisions may be rationale but can also be irrational. Decision making involves evaluating arguments for and against a solution to a problem. Tools can support decision making by providing solutions and justifications for the designer to confirm or choose between.

Decision Recording Decisions recording is the documentation of the design process. It is a separate activity from decision making. When, for example, a decision is taken but not recorded the decision becomes an assumption. Assumptions, of course, are undesirable because implicit design decisions hinder maintenance. Documenting the design process with explicit decisions taken provides an essential record for when the design needs to be modified because design decisions might have to be reviewed in light of the modifications at hand. In this respect tools should facilitate documenting the decisions *at the time* they are made and before the rationale behind them is lost.

Exploration Exploration is the process of examining alternatives to a design problem. Design problems are often 'solved' with the first satisficing solution found because of their sheer complexity [13, p. 36]. What must be stressed here is that the design process should be concerned with the effectiveness of a solution and not with its efficiency: the designer should not be concerned with finding *the* solution but analyzing a range of solutions. Design tools should allow the designer to backtrack in his, or her, design process and therefore permit the designer to freely explore alternative solutions to a problem without the cognitive overhead of remembering a found satisficing solution.

Construction Construction is act of 'doing' design and progressing from specifications to solutions. Construction is rarely an argumented activity and therefore is difficult to record. Design tools can provide the 'building blocks' for a given design domain.

Argumentation Argumentation is the counterpart to construction [12]. Argumentation is the act of deliberating about the state of a design and correcting any perceived undesirable elements. Tools can provide elements of a design to analyze by applying rules and heuristics to a design and detecting anomalies.

Iteration Design is an iterative process between construction and argumentation (Morch [12] proposes the term *reflection in action* and Simon [21] the term *generator-test cycle*). Both construction and argumentation are interleaved and tool support should reflect this cyclic nature.

2 The Literate Data Modelling Paradigm

Literate programming is an approach to programming proposed by D.E. Knuth [7,1] that promotes interleaving, in the same document, both the source code of a program and a full account of the rationale that went into constructing the program. Literate programs should be as readable, and interesting, as a piece of literature.

Database modelling is not the same activity as programming but both are design activities and share common characteristics such as choosing between alternatives, exploration of possible representations and testing whether the result of the design activity corresponds (or satisfies) the given requirements.

The literate data modelling paradigm which we present here, as with literate programming, considers the documentation of the design process (using deliberation schemas) as important as the results of the process (the design products).

In this section we will first present some assumptions and limitations of our approach. We will briefly justify why an algorithmic approach is not always sufficient. We will then present deliberation schemas as a structure for the design process, first through an example, then in more detail.

2.1 Assumptions

We will be making certain assumptions about design products and the design process and limiting our approach to only certain aspects of both.

First, we will be limiting ourselves to examining only the data analysis and data modelling phases of the design process.[1]

We will also be assuming that the modelling process is undertaken using the relational data model.

We will be assuming that the modelling process is punctuated with remarkable situations (*cas remarquables* [9,10]). A remarkable situation reflects a specific state of the evolving design that necessitates the intervention of an analyst and/or designer. We will be using the term deliberation to describe this intervention.

2.2 Why not an algorithmic approach?

Relational data modelling (or logical data modelling) is usually seen as a task that can be aided with algorithms — for example by algorithm that decomposes a scheme into 3NF. However the algorithmic solution to the decomposition problem is not without its problems as we will show in the next few paragraphs.

The first is that complete and coherent data is necessary for the algorithms to give meaningful results. If the analysis phase specifies, for example, that the functional dependencies $a \rightarrow c$ and $ab \rightarrow c$ hold, then any algorithm will arbitrarily reject $ab \rightarrow c$ as not being elementary. Likewise, if the functional dependencies $a \rightarrow b$ and $b \rightarrow c$ are specified then any algorithm will generate $a \rightarrow c$. In the first case the algorithm retracts a dependency, in the second it asserts a dependency. Checking for incompleteness and incoherence could be tasks within the analysis phase but we believe that this is rarely possible or even desirable. We believe it is very difficult for analysts to acquire *all* the information necessary before the modelling phase starts. If the analysis phase is undertaken by separate groups then it is preferable not to try and reduce the viewpoints to a single one, if in fact the 'reality' is viewed differently. We believe that the study of the dependencies during the modelling phase reveals such cases of incompleteness and inconsistency. The designer — or the algorithm — should not arbitrarily decide on correcting any incoherence or inconsistency; instead the designer should consult with the analysts responsible for the analysis phase for clarification.

Another problem with the algorithmic approach is that, usually, only one result is given when in fact a number of equivalent results are possible. This is often true for algorithms that calculate decompositions that may return only one decomposition when in fact the are a number possible. The choice of decomposition should also be under the control of the designer and not arbitrarily chosen by an algorithm.

[1]we hope to show, however, that the paradigm could be applied to all phases and elements of design process

Algorithms are frequently simple functions and as such can be viewed as 'black-boxes' that given data for input will return a result as output. The problem here is that the designer using such an algorithm is not given any feedback or justification about the algorithm's result.

A final problem we will mention here is that, for some algorithmic problems, there exists no solution. For example, it is not always possible for a scheme to satisfy a lossless join and dependency-preserving BCNF, as we will see in Section 2.3.1.

2.3 Deliberation schemas

The literate data modelling paradigm prescribes using deliberation schemas as a structuring method for decision-taking and documentation.

Deliberation schemas provide a default argumentation structure about some aspect of the evolving design. They are based on Toulmin's work on argument patterns [23] and the IBIS method for structuring the design process as a conversation [8].

The principle elements of deliberation schemas are issues, positions, and arguments.

Issue Issues provide a focus of concern during the design process. Issues can arise from reviewing design artifacts with respect to certain criteria, rules, norms or heuristics. Issues can also be raised by the design process itself; for example, selecting a position to an issue might cause an issue to be raised.

Position A position is a candidate response to an issue. Positions will often be mutually exclusive (but is not necessary).

Argument Arguments are justifications that support and/or object to positions. Arguments establish claims [23] for and against positions. A single argument can support one position yet object to another.

2.3.1 An example deliberation schema

We will see — through a simple design scenario — how the structure of issues, positions and arguments can help structure the decisions a designer might be confronted with. Take the following database scheme[2] that has the functional dependency $city, street \rightarrow postalCode$ defined on it:

Addresses(city, streetpostalCode), for all tuples of Addresses (c s p), city c has a building with street address s and p is the postal code of for that address in that city.

The single key for the scheme is (city street) and the scheme is in Boyce-Codd Normal Form (BCNF) because all the elementary functional dependencies are those in which a key functionally determines one or more of the attributes.

Now imagine that the functional dependency $postalCode \rightarrow city$ also holds for *Addresses*. The set of keys for *Addresses* becomes (street city) and (street postalCode). The scheme *Addresses* is no longer in BCNF because the left-hand-side of $postalCode \rightarrow city$ is not a key of *Addresses*. *Addresses*, however is in 3NF because *city* is a prime attribute. Decomposing the scheme into BCNF

[2]adapted from [24]

StreetCodes(street, postalCode) and *CityCodes(postalCode//city)*[3] does not preserve dependencies because *city, street* $\rightarrow$ *postalCode* is not implied by the projected dependencies.

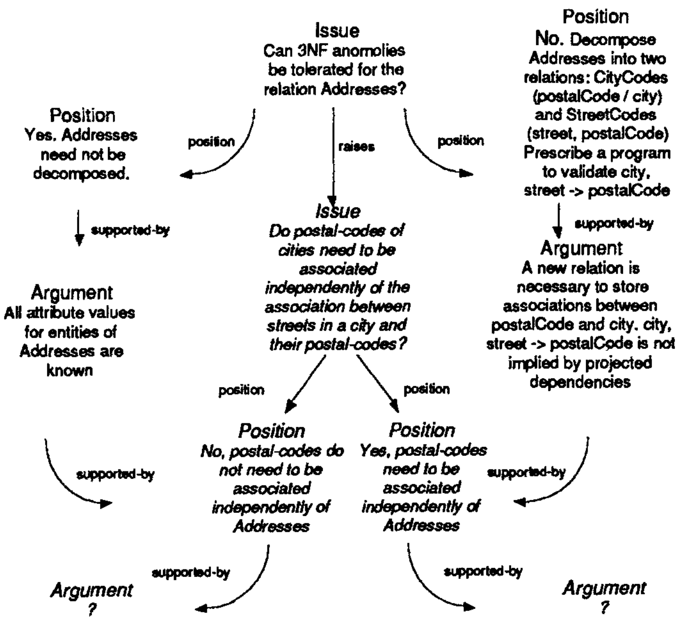

Issue
Can 3NF anomolies be tolerated for the relation Addresses?
Position
Yes. Addresses need not be decomposed.
position
raises
position
Position
No. Decompose Addresses into two relations: CityCodes (postalCode / city) and StreetCodes (street, postalCode) Prescribe a program to validate city, street -> postalCode
supported-by
Issue
Do postal-codes of cities need to be associated independently of the association between streets in a city and their postal-codes?
supported-by
Argument
All attribute values for entities of Addresses are known
Argument
A new relation is necessary to store associations between postalCode and city. city, street -> postalCode is not implied by projected dependencies
position
position
Position
No, postal-codes do not need to be associated independently of Addresses
Position
Yes, postal-codes need to be associated independently of Addresses
supported-by
supported-by
supported-by
supported-by
Argument
?
Argument
?

Figure 1: An initial deliberation schema

Figure 1 represents an initial deliberation schema for the designer and analyst.[4] Issue, positions and arguments are represented as nodes. The nodes in *italics* are those concerning the analyst, those in plain concern the designer. We will first examine the deliberation concerning the designer.

The designer's issue in this situation is whether the anomalies of 3NF can be tolerated for the relation *Addresses*. Two positions respond to the issue: the first tolerates the anomalies of 3NF and proposes to not decompose *Addresses*; the second position proposes decomposing *Addresses* into a non dependency-preserving BCNF and prescribing a validation method for *city, street* $\rightarrow$ *postalCode*. The position to accept 3NF anomalies is justified by the argument that all attribute values for entities of *Addresses* are known. The second position is justified by the argument that a new relation is necessary to store associations between *postalCode* and *city*.

Both arguments, however, are supported by positions that respond to an analysis issue, and as such the designer must suspend deciding on his/her position until s/he can get a confirmed position from the analyst on the issue of whether postal-codes need to be associated independently of addresses.

[3]non prime attributes, if the relation has any are written to the right of the double-slash (//), if the relation accepts more than one key then they are separated by a single-slash (/)

[4]by analyst we refer to a person (or group of people) that are responsible for communicating with end-users about application domain requirements and by designer as a person (or group of people) that are responsible for developing computer models and systems respecting specifications produced by analysts. Analyst and designer *could* be the same person or group of people

The analyst's issue is to elicit from end-uses whether or not postal-codes need to be associated independently of addresses. The two positions responding to this issue are simply the positive and negative responses. No formal *a priori* justifications can be given for either position (as these would concern the application domain) so the initial deliberation schema does not propose any default arguments.

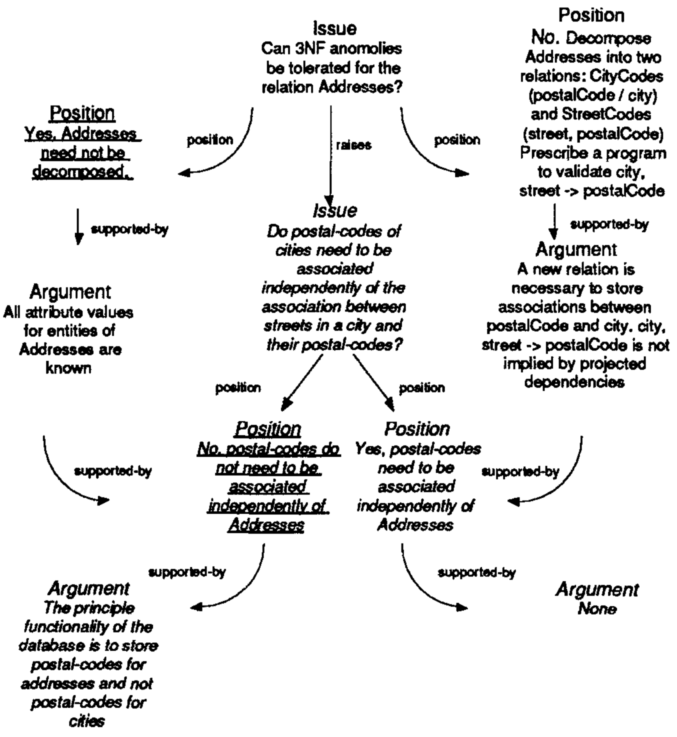

Issue
Can 3NF anomolies be tolerated for the relation Addresses?
Position
No. Decompose Addresses into two relations: CityCodes (postalCode / city) and StreetCodes (street, postalCode) Prescribe a program to validate city, street -> postalCode
Position
Yes. Addresses need not be decomposed.
position
raises
position
supported-by
Issue
Do postal-codes of cities need to be associated independently of the association between streets in a city and their postal-codes?
supported-by
Argument
A new relation is necessary to store associations between postalCode and city. city, street -> postalCode is not implied by projected dependencies
Argument
All attribute values for entities of Addresses are known
position
position
Position
No. postal-codes do not need to be associated independently of Addresses
Position
Yes, postal-codes need to be associated independently of Addresses
supported-by
supported-by
supported-by
supported-by
Argument
The principle functionality of the database is to store postal-codes for addresses and not postal-codes for cities
Argument
None

Figure 2: Deliberation in favour of not associating postal-codes and addresses independently

We will see how the deliberation, in this simple example, can go one of two ways. First, let's imagine that the analysts elicit from the end-users that: *The principle functionality of the database is to store postal-codes for addresses and not postal-codes for cities.* This is clearly support for the position that postal-codes do *not* need to be associated independently from addresses and is recorded within the deliberation schema as a supporting argument for this position. The analyst deliberates in favour of this position. The designers now has support for the argument in favour of not decomposing the *Addresses* relation and may deliberate in favour of this position. Figure 2 resumes this deliberation. Note the end-user argument and the selected (underlined) positions.

We can easily imagine a different argument the analysts might elicit from the end-users: *The data given by the Post Office concerns only postal-codes and cities.* This argument supports the position for that postal-codes need to be associated separately from addresses. Imagine that the analysts deliberates in favour of this position. The designer now has support for the argument in favour of decomposing the *Addresses* relation. Figure 3 resumes this deliberation. Note the end-user argument and the selected (underlined) positions.

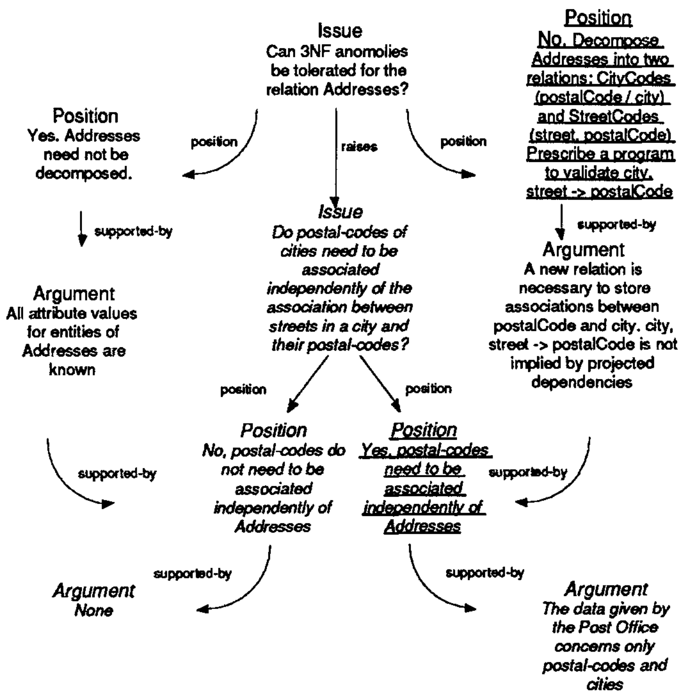

Issue
Can 3NF anomolies be tolerated for the relation Addresses?
Position
Yes. Addresses need not be decomposed.
position
raises
position
Position
No. Decompose Addresses into two relations: CityCodes (postalCode / city) and StreetCodes (street, postalCode). Prescribe a program to validate city, street -> postalCode
supported-by
Argument
All attribute values for entities of Addresses are known
Issue
Do postal-codes of cities need to be associated independently of the association between streets in a city and their postal-codes?
supported-by
Argument
A new relation is necessary to store associations between postalCode and city. city, street -> postalCode is not implied by projected dependencies
position
position
supported-by
Position
No, postal-codes do not need to be associated independently of Addresses
Position
Yes, postal-codes need to be associated independently of Addresses
supported-by
supported-by
Argument
None
supported-by
Argument
The data given by the Post Office concerns only postal-codes and cities

Figure 3: Deliberation in favour of independently associating postal-codes and addresses

2.4 A model for deliberation schemas

We will now propose a model for deliberation schemas. We will present their static aspects first, followed by definitions for their dynamic triggering.

2.4.1 Static aspects

Figure 4 illustrates the associations permitted between elements of deliberation schemas. Issues, positions and arguments are nodes and are associated with labeled arcs. A deliberation schema is therefore a directed graph consisting of typed nodes (issue, position or argument).

We will now examine the properties of each node-type:

Issue An issue has properties concerning its audience, expression, and resolution. The **audience** property indicates whether the issue is addressed at analysts or designers. The **expression** property indicates the focus of concern in natural language, usually in the form of a question. The **artifacts** property lists the

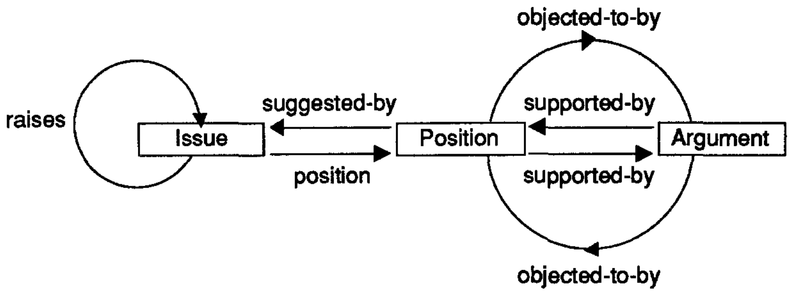


Figure 4: A model for deliberation schemas

set of artifacts the issue is concerned with. They provide the 'evidence' for the issue. The **resolved** property is initially given the value 'false' but is set to 'true' (by the analyst or designer) when the issue is considered resolved.

Position A position has three properties: its expression, whether or not it is selected and its audience. The **expression** property is the position expressed in natural language, usually as an assertion. The **selected** property indicates whether the position has been selected and is initially set to false but is set to true by the analyst or designer when the position in chosen. The **audience** property indicates whether the issue is addressed at analysts or designers.

Argument An argument has two properties: its expression and its audience. The **expression** indicates a justification and the **audience** indicates whether the argument is addressed at designers or analysts.

2.4.2 Dynamic aspects

Section 2.4.1 described a structure for storing deliberation schema in a CASE-tool dictionary. It did not, however, describe how, or when, a deliberation schema could automatically be created.

A deliberation schema need to be created whenever a certain state of evolving model reflects the need for designer and/or analyst intervention. This state can be specified as a condition. Let's call this condition the *triggering condition*. In the example presented in Section 2.3.1 the triggering condition would be a relation being in 3NF yet not decomposable into BCNF.

Normally, the triggering condition would have to be tested each time the evolving model is changed to ensure that the set of deliberation schemas created would be up-to-date. To reduce this costly operation a *triggering range* can define a set of modelling primitives that necessitate testing the triggering condition. We saw in Section 2.3.1 how adding a functional dependency could cause a triggering condition to be satisfied. This primitive would be an element of the deliberation schema's triggering range.

2.5 Advantages of deliberation schemas

Let's examine some of the advantages of using deliberation schemas during the design process:

1. All positions to an issue are explicit and the designer (or analyst) is encouraged to explore all the positions before choosing one over the other.

2. Arguments for choosing between positions are explicit and place the designer on clear ground for decision-making.

3. Rationale for position selection is clear. By choosing a position the designer (or analyst) is implicitly accepting its supporting arguments and refuting its objecting arguments. The decision *not* to choose a position implicitly means that its supporting arguments (if any) do not play an important enough role for its selection and/or its arguments objecting to it are important enough for its rejection.

4. Separation of responsibilities, since issues regarding analysis and design are separate. In the example the analyst's task is clear: elicit from the end-users whether city postal-codes need to be associated independently of addresses. The designer's issue is quite separate: whether or not to tolerate the *Addresses* relation in 3NF. Issues addressed to analysts concern aspects and requirements of the application domain whereas issues addressed to designers are more technical in nature and refer to properties of the evolving model. The responsibilities of analysts and designers are clearly distinguished. Designers can not make short-cuts by accepting a position without justification by analysts (if the deliberation schema requires it; i.e. a designer argument is supported by a analyst's position). In this case a deliberation schema can be seen as prompts for the designer to collaborate with analysts and obtain additional information from them before proceeding with the design.

5. Issue precedence is inherent in the schema. In the example the issue *Do postal-codes of cities need to be associated independently of the association between streets in a city and their postal codes?* needs to be resolved before the issue *can 3NF be tolerated for Addresses?* since the arguments to the second issue can not be supported before positions have been taken on the first.

6. Documentation becomes an integral part of the design process. The designers (and analysts) are facilitated and encouraged to structure their design process using the deliberation schemas. Documentation of information systems is rarely undertaken during the design process itself. Documentation is usually considered at the end of the process once a stable system is obtained. The problem with documenting 'after-the-fact' is that most of the rationale behind the decisions and trade-offs taken during the design is not available any more. Yet good documentation is critical to maintaining a system in response to changing requirements. Deliberation schemas facilitate documenting *during* the process of actually taking design decisions. This allows the design to be reliably associated with the complete rationale that went into its construction.

7. Design enrichment can be accommodated. In the example it is specified that a program must be written to ensure a functional dependency not validated by the decomposition. This information becomes part of the documentation of the modelling phase and a requirement for the implementation phase. This is an important because most design dictionaries do not allow this kind of supplementary information during the design phase. Note that the supplementary requirement will not only specify that such a program is necessary but will include the *justification* for such a program in terms of the application domain and modelling constraints.

8. Common documentation format. Deliberations schemas provide a common structure for recording rationale of decisions taken during the design process.

3 NelleN: an Implementation Framework

Most CASE-tools implement a data dictionary that stores the *results* of the development process. We propose augmenting the data dictionary with deliberation schemas that record the *process* of reaching those results. NelleN is a framework for implementing CASE-tools with such an augmented dictionary.

3.1 The NelleN Framework

Implementation frameworks are foundation architectures for building computer applications. They are based on two principles; the first that a class of programs can share a core set of code that should not be re-written each time a program of this class is written, the second that most code particular to a specific application can be written as subclasses of the core code. One such framework is the MacApp framework [20] for developing Apple Macintosh applications; another is the Model-View-Controller (MVC)[6] framework for implementing graphical user-interfaces.

Reflecting these two basic principles, the NelleN framework is divided into two parts. The first, the NelleN kernel, is a core set of abstract classes that supports deliberation schemas. The second part of the framework consists of a subclassing mechanism to extend the framework with concrete classes for specific types of deliberation schemas.

The NelleN framework has been implemented in Smalltalk. Smalltalk is an object-oriented programming language and its subclassing mechanism is ideally suited for implementing a framework. Smalltalk has already proved an ideal vehicle for the MVC framework — in fact its programming environment is written as an extension of the MVC framework. Smalltalk also has the advantage of a large and stable class library. The extensive collections class hierarchy, for example, is a valuable aid to implementing modelling algorithms that frequently need sets for their implementation.

3.1.1 The NelleN kernel

The NelleN kernel provides the general functionality for triggering and storing deliberation schemas.

The NelleN kernel is a set of Smalltalk class hierarchies. These class hierarchies are divided into three categories: the algorithm kernel, the deliberation schema kernel, and the deliberation triggering kernel.

Figure 5 represents the current structure and status of the NelleN kernel. Items in a `typewriter` typeface are classes. All links indicate the subclass relationship. The roots of each partial class hierarchy are part of the standard Smalltalk implementation.[5]

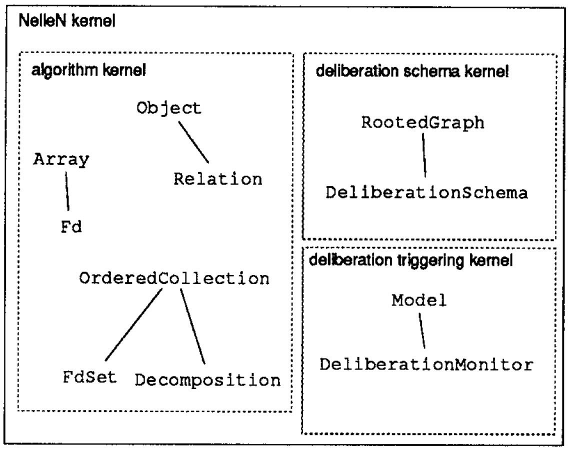


Figure 5: The NelleN kernel

The algorithm kernel

The algorithm kernel is a set of classes that implement the data structures necessary to store the data model and calculate its properties. Smalltalk classes are used to represent modelling entities. For example, the class `FdSet` is a subclass of the Smalltalk class `OrderedCollection` that accepts elements of class `Fd`. Methods implement algorithms that can be performed on modelling entities. For example, the class `FdSet` provides a method named `minElementaryClosure` that returns the minimal elementary closure of a set of functional dependencies.

The deliberation schema kernel

We saw in Figure 4 that deliberation schemas can be represented by directed graphs. The basic functionality of deliberation schemas is therefore implemented by an abstract class, `DeliberationSchema`, a subclass of `RootedGraph`. Being an abstract class, `DeliberationSchema` does not provide protocols for creating instances. Concrete subclasses of `DeliberationSchema` handle this task as we will see in Section 3.1.2.

[5]the `RootedGraph` class are not a standard part of the Smalltalk hierarchy. We have used the public domain graph classes developed by Mario Wolczko of the University of Manchester

The deliberation triggering kernel

The deliberation triggering kernel is roughly based on the Model-View-Controller mechanism (MVC). The MVC-triad allows the model (or application) to be developed (and maintained) independently of its interface (a common requirement for highly interactive applications). Views and controllers in MVC are dependent on their model. The model itself does not 'know' about the views that are dependent on it but simply broadcasts messages to them if an aspect of it changes. Views are updated using this mechanism.

The deliberation triggering mechanism is similar because we want the data model dictionary and the operations that can be performed on it to be separate from the triggering of deliberation schemas. The benefit of this approach is that deliberation schema triggering becomes configurable (any number of deliberation schema 'types' can be 'installed') just as in the MVC paradigm (any number of arbitrary views can be displayed, independent of the model).

The deliberation triggering mechanism is achieved by the abstract class `DeliberationMonitor`. The `DeliberationMonitor` class is responsible for 'registering' the instances of its subclasses as dependents of a data model. Subclasses of `DeliberationMonitor` are 'paired' with a concrete subclass of class `DeliberationSchema` (the class of deliberation schema it raises). Using the Smalltalk update mechanism subclasses of `DeliberationMonitor` are responsible for creating instances of issues to be stored in the dictionary.

3.1.2 Extending the NelleN framework: concrete subclasses

The NelleN kernel provides only the general functionality for creating and storing deliberation schemas. Deliberation schema types (corresponding to a remarkable situation type) are implemented by writing two concrete subclasses (somewhat like implementing a view-controller pair in the MVC framework): the first a concrete subclass of `DeliberationSchema`, the second a concrete subclass of `DeliberationMonitor`. Figure 6 shows where in the framework a deliberation schema pair (underlined in the figure) are placed. The first implements the structure of the issue, the second the conditions for creating an instance of that deliberation schema type.

Subclassing `DeliberationMonitor`

Concrete subclasses of `DeliberationMonitor` implement the triggering and range conditions for creating instances of a deliberation schema type.

Two methods need be implemented (the dependency methods are inherited from `DeliberationMonitor`). The first is an instance creation class method that creates an instance of its class and assigns itself as a dependent of a model. This method must respect the protocol `on:aModel` and typically contains only a few lines of code.

The second method is responsible for determining whether a certain state holds true for the data model and (if so) creating an instance of its 'paired' deliberation schema type. This method must have the selector `update:anAspect`. This is a typical Smalltalk keyword selector with the keyword `update` and the argument `anAspect`. This method implements the response necessary during the Smalltalk update mechanism. If, for example, a functional dependency is added to the data

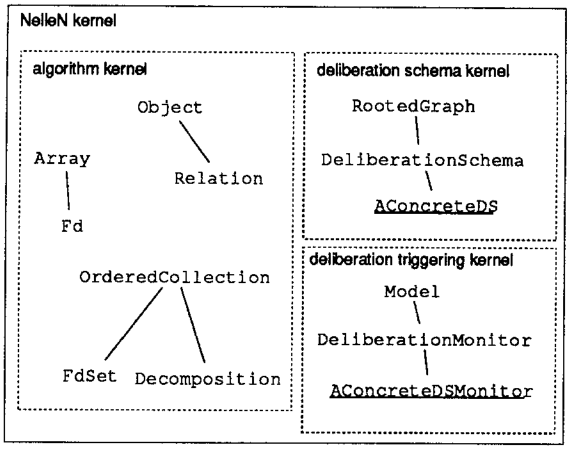


Figure 6: Extending the NelleN Framework

model, the method implementing this modelling primitive will contain a line of code **`self update:#fdAdded`**. This means that each dependent (in this case instances of subclasses of **`DeliberationMonitor`**) will receive the message **`update:anAspect`** 'broadcasted' by the data modelling primitive. The dependent receiving this message can determine whether it is concerned by this message by testing the **`:anAspect`** argument. If for example, the range of the deliberation schema we want to implement includes the **`add functional dependency`** primitive then the **`update:anAspect`** method would test if **`:anAspect`** equals the symbol:**`#fdAdded`** and then test to see if its condition holds. The condition is tested by asking the model (in this case the data model) for its elements. If the condition is found to hold true on the data model then the method creates the deliberation schema by sending an instance creation method to its 'paired' class and then sending a message to the model asking for the deliberation schema to be stored.

Subclassing `DeliberationSchema`

Subclasses of `DeliberationSchema` receive instance creation messages from its paired monitor class. They are responsible for creating a deliberation schema graphs similar to the example in Figure 1.

All the methods for actually constructing the deliberation schema graph, testing, accessing and displaying the graph are part of the **`DeliberationSchema`** class and do not need to be re-written for each subclass because of the code inheritance mechanism. Subclasses must specify the graph that represents its deliberation schema, according to the model outlined in Section 4.

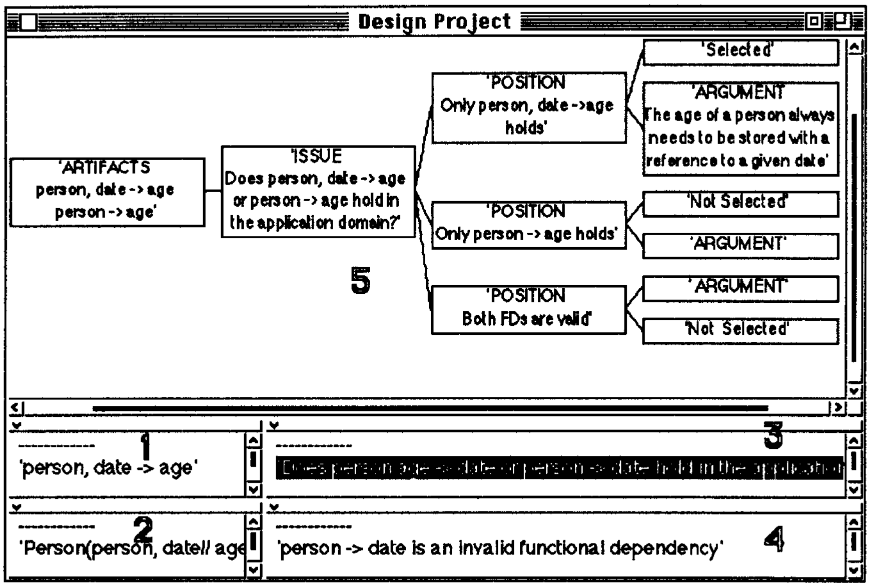
3.2 A prototype using NelleN

We will now briefly present a simple example of the type of modelling tool that can be built with using the NelleN framework. It does not in any way pretend to be a complete tool but simply to demonstrate the feasibility of the framework.

The example we propose here uses only one simplified concrete deliberation schema class: one that queries whether a pair of functional dependencies are contradictory or not.

Figure 7 illustrates this prototype. [6] The example shown here displays the deliberation schema graph slightly differently from the one described in Figure 4. Here we consider the artifacts and selected properties as graph nodes.

The prototype uses both the MVC framework for its interface and the NelleN framework for its deliberation schema. Two hundred lines of Smalltalk code were necessary to build this (admittedly small) example, excluding the frameworks.


Design Project
'ARTIFACTS person, date -> age person -> age'
'ISSUE Does person, date -> age or person -> age hold in the application domain?'
'POSITION Only person, date ->age holds'
'POSITION Only person -> age holds'
'POSITION Both FDs are valid'
'Selected'
'ARGUMENT The age of a person always needs to be stored with a reference to a given date'
'Not Selected'
'ARGUMENT'
'ARGUMENT'
'Not Selected'
5
1
3
2
4

'person, date -> age'

Does person, age -> date or person -> date hold in the applicatio

'Person(person, date// age

'person -> date is an invalid functional dependency'

Figure 7: A example modelling tool using the NelleN Framework

The interface chosen for the prototype is one that is often used in Smalltalk applications: a browser interface. The browsing window consists of a number of views, or panes, which we will briefly examine in turn:

view 1: the list of functional dependencies

view 2: the list of relations forming the decomposition

view 3: the list of issues addressed to the analyst. If an item in the list is selected (like in Figure 7) the partial deliberation schema (corresponding to the subgraph that concerns the designer) is displayed in the larger view (5)

view 4: the list of issues addressed to the designer. If an issue in the list is selected it is displayed in the larger view (5).

[6] the windows panes have been numbered for demonstration purposes

view 5: this view graphically displays the last selected issue of views (3) and (4). The layout and display are automatic[7].

4 Related Work

We will briefly review other research work that is related to our approach.

The research closest to our approach is probably the work of Rolland [18,22,3]. Rolland proposes 'representation-triplets' consisting of a situation, a decision and an action to guide the designer in his, or her, work. The proposed situations are similar to deliberation schema triggering conditions because they both indicate a significant state of the evolving design. A 'decision' corresponds to a choice made by the designer and can easily be compared with the selecting of a position in our approach. An 'action' consists of the transformations performed resulting from a decision and resemble the modelling positions we propose. Rolland, however proposes a fully fledged CASE-tool (ALECSI) based on an expert system approach, while we propose an extendable framework for implementing CASE-tools.

Conklin and Begeman have demonstrated with gIBIS [5] the feasibility of an issue-based tool. gIBIS is a hypertext system for capturing the rationale behind early design decisions and has proved useful in the domain of CSCW (Computer Supported Cooperative Work).

Potts and Bruns [16,15] have worked on the importance of documenting the decision process in software engineering by proposing a generic model of deliberation. Their model is also based on Rittel's IBIS method.

Finally, Rätz, Lusti and Glaubauf [17] have designed and implemented an ITS (Intelligent Tutoring System) that tutors students on the task of data normalization. They propose 'psychologically valid' algorithms for data normalization that are closer to how designers actually reason about normalization. Their diagnostic model, for when errors are made, is similar to our approach of positions and arguments responding to issues.

5 Conclusions

We have proposed the literate data modelling paradigm that takes into account the iterative nature of the data modelling design process. We have argued that it can structure the complex design process by integrating deliberation and documentation.

We have also presented an extendable implementation framework, NelleN, that implements the abstract functionality of deliberation schemas. We have shown that the framework can easily be extended by using a subclassing mechanism. A prototype has been presented that uses the framework.

The research we have presented here is on-going. Research efforts currently being pursued are identifying and classifying the types of deliberation schemas that can be encountered during the modelling process and extending the NelleN framework

[7]the grapher classes developed by Mario Wolczko of the University of Manchester perform the automatic layout and displaying of the graph

with them. We are also considering implementing a meta-CASE-tool that would configure the NelleN framework to a given modelling method.

References

[1] BENTLEY, J. Literate programming. *Communications of the ACM 29*, 5 (May 1986), 364 - 369.

[2] BUBENKO, JR, J. A. Information system methodologies — a research view. In *Information Systems Design Methodologies: Improving the Practice* (Amsterdam, 1986), T. Olle, H. Sol, and A. Verijn-Stuart, Eds., IFIP, North-Holland, pp. 289 - 318.

[3] CAUVET, C., PROIX, C., AND ROLLAND, C. ALECSI: an expert system for requirements engineering. In *Advanced Information Systems Engineering, CAiSE'91* (1991), R. Anderson, J. Bubenko, and A. Sø lvberg, Eds., vol. 498 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 31–49.

[4] CERI, S. Methodology and tools for data base design. In *Methodology and Tools for Data Base Design*, S. Ceri, Ed. North-Holland, 1983, pp. 1 - 6.

[5] CONKLIN, J., AND BEGEMAN, M. gIBIS: a hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems 6*, 4 (1988), 303 - 331.

[6] GOLDBERG, A. *Smalltalk-80: the Interactive Programming Environment.* Addison-Wesley, 1984.

[7] KNUTH, D. E. Literate programming. *Computer Journal 27*, 2 (May 1984), 97 - 111.

[8] KUNZ, W., AND RITTEL, H. Issues as elements of information systems. Working Paper 131, Institute of Urban and Regional Development, University of California, 1970.

[9] LÉONARD, M. *Structure des Bases de Données.* Dunod, 1988.

[10] LÉONARD, M. *Database Structures.* Macmillan, 1992. In press.

[11] MACDONALD, I. Automating the information engineering methodology with the Information Engineering Facility. In *Computerized Assistance During the Information Systems Life Cycle* (1988), T. Olle, A. Verrijn-Stuart, and L. Bhabuta, Eds., North-Holland, pp. 337 - 373.

[12] MORCH, A. JANUS: Basic concepts and sample dialog. In *CHI'91, ACM Conference on Human Factors in Computing Systems — Reaching Through Technology* (1991), S. Robertson, G. Olson, and J. Olson, Eds., ACM, pp. 457–458.

[13] NEWELL, A., AND SIMON, H. *Human Problem Solving.* Prentice-Hall, 1972.

[14] PETERS, L. *Advanced Structured Analysis and Design.* Prentice-Hall, 1988.

[15] POTTS, C. A generic model for representing design methods. In *11th International Conference on Software Engineering* (1989), pp. 217 - 226.

[16] POTTS, C., AND BRUNS, G. Recording the reasons for design decisions. In *10th International Conference on Software Engineering* (1988), Computer Society Press, pp. 418 – 427.

[17] RÄTZ, T., LUSTI, M., AND GLAUBAUF, M. An intelligent tutoring system for database normalization. WWZ-discussion papers, Universität Basel, WWZ, 1991.

[18] ROLLAND, C., AND PROIX, C. Vers une automisation des processus de conception par les outils. In *Autour et à l'Entour de Merise. Les Méthodes de Conception en Perspective* (1991), CERAM, AFCET, pp. 271 - 286.

[19] ROLLAND, C., AND RICHARD, C. The Remora methodology for systems design and management. In *Information Systems Design Methodologies* (1982), T. Olle, H. Sol, and A. Verijn-Stuart, Eds., North-Holland, pp. 369 – 426.

[20] SCHMUCKER, K. *Object-Oriented Programming for the Macintosh.* Hayden, 1986.

[21] SIMON, H. *The Sciences of the Artificial*, 2 ed. MIT Press, 1981.

[22] SOUVEYET, C., AND ROLLAND, C. Correction of conceptual schemas. In *Advanced Information Systems Engineering, CAiSE'90* (1990), B. Steinholtz, A. Solvberg, and L. Bergman, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 152 - 174.

[23] TOULMIN, S. *The Uses of Argument.* Cambridge University Press, 1958.

[24] ULLMAN, J. *Database Systems*, 2 ed. Computer Science Press, 1983.