Declarative Programming and Clinical Medicine

On the Use of Gisela in the MedView Project

Olof Torgersson

Department of Computing Science, Chalmers University of Technology
and Göteborg University, S-412 96 Göteborg, Sweden
oloft@cs.chalmers.se

Abstract. In 1995, the MedView project, based on a co-operation between computing science and clinical medicine was initiated. The overall aim of the project is to develop models, methods, and tools to support clinicians in their diagnostic work. Today, the system is in daily use at several clinics and the knowledge base created contains more than 2000 examination records from the involved clinics. Knowledge representation and reasoning within MedView uses a declarative model based on a theory of definitions. In order to be able to model knowledge declaratively and integrate reasoning into applications with GUIs a framework for definitional programming has been developed. We give an overview of the project and of how declarative programming techniques are integrated with industrial strength object-oriented programming tools to facilitate the development of real-world applications.
Keywords: Definitional programming, Clinical Medicine, Integration with GUIs and Objective-C.

1 Introduction

Diagnostic work and clinical decision-making are central items in every field of medical practice, where clinical experience, knowledge and judgement are the cornerstones of health care management. In order to achieve increased competence, the clinician is confronted with complex information that needs to be analysed. Accordingly, the clinician needs tools to improve analysis and visualization of data in the diagnostic and learning processes.

The traditional paper record used in medicine does not store information in a manner that makes it easy to learn from the huge amounts of information collected over time. Not only does it require that someone manually reads and organizes the stored information, but even if the records are read the information stored within them is not sufficiently organized and formalized to form a basis for a general analysis. Unfortunately most computerized systems developed today to replace the paper record share the same problem. Systems are designed for storage and transportation of data, and to simplify administrative tasks.

To remedy these problems systems must be designed which are from the beginning focused on knowledge representation and reasoning, areas where declarative programming tools typically are a natural choice.

In 1995, the Medview project [1], based on a co-operation between computing science and oral medicine, was initiated. The overall goal of the project is to develop models, methods, and tools to support clinicians in their diagnostic work. The project is centered around the question: how can computing technology be used to handle clinical information in everyday work such that clinicians more systematically can learn from their gathered clinical data? That is, how can the chain "formalize-collect-view-analyse-learn" be understood and implemented in the area of clinical medicine.

The basis for formalizing clinical knowledge within MedView is a uniform definitional model suitable for automated reasoning in a computerized system. At the same time, the model is simple enough to have an obvious intuitive reading to the involved clinicians needing no further explanation. To implement the declarative formalized model of clinical knowledge suitable programming tools are needed. Since interacting with users is another very important part of a system for use in a clinical setting we also need programming tools suitable for building graphical user interfaces. The solution taken in the MedView project is to use declarative programming techniques and state-of-the-art object-oriented programming tools in concert using each for the task it performs best. To ensure a smooth integration a framework for definitional programming has been developed that can be seamlessly integrated into applications of various kinds.

The rest of this paper is organized as follows. In Sect. 2 we give some background on definitional programming and the MedView project. Section 3 provides an overview of Gisela, a new framework for definitional programming. In Sect. 4 we describe how Gisela is used in the MedView project. Finally, Sect. 5 gives some notes on related work and possible future directions.

2 Background

2.1 Definitional Programming

Declarative programming comes in many flavors. Common to most declarative paradigms is the concept of a definition. Function definitions are given, predicates are defined etc. However, focus is on what we define, on the functions and predicates respectively. *Definitional programming* is an approach to declarative programming where the definition is the basic notion, that is, focus is on the definitions themselves, not on what we define.

The definitional programming language GCLA[1] [3, 2, 12] was developed as a tool suitable for the design and implementation of knowledge-based systems. In GCLA, programs are regarded as instances of a specific class of definitions, the *partial inductive definitions* (PID) [9]. The definitions of GCLA consist of a number of definitional clauses

$$a = A. \ ,$$

[1] To be pronounced Gisela.

where an *atom* a is defined in terms of a *condition* A. The most important operation on definitions is the *definiens* operation, $D(a)$, which gives all the conditions defining the atom a in the definition D.

In GCLA, control information is completely separated from domain information. This is achieved by using two kinds of definitions when constructing programs: The (object) *definition* and the *rule definition*. The intention is that the definition by itself gives a purely declarative description of the problem domain, while the rule definition contains the procedural knowledge needed to perform computations.

From a programming point of view, GCLA is basically a logic programming language, sharing syntax, logical variables, depth-first-search and backtracking with Prolog. The explicit representation of control makes definitional programming a more low-level approach, compared to other declarative programming paradigms, and for most programs, the programmer must be aware of control issues.

Definitional programming in the form of GCLA has been used in a number of applications including construction planning, diagnosis of telecommunication equipment, and music theory. It has also been used to implement a definitional approach to the combination of functional and logic programming [14].

2.2 The MedView Project

Essentially, the MedView project can be divided into two sub-problems: knowledge representation and reasoning, and development of applications for gathering and exploring clinical data. A first strategic decision was to not try to build yet another electronic medical record system, but to focus on knowledge gathering and analysis based on a formal description of the concept "examination". Another to build applied software, for use in the clinical setting, in parallel with the development of theory and implementation of tools for knowledge representation and reasoning. This led to the following things to be done, approximately in the given order:

1. Provide a formal framework and methodology to be used.
2. Formalize the knowledge to be gathered based on this methodology in a close cooperation between odontologists and computer scientists.
3. Develop tools for entering the information gathered at an examination into the knowledge base directly in the examination room.
4. Develop tools for viewing the contents of the knowledge base, both for use in the examination room and later for retrospective studies.
5. Develop tools for analyzing and exploring the knowledge base and for adding concepts built on top of the basic formal method.

Knowledge Representation. In MedView clinical data is seen as definitions of clinical terms. Abstract clinical concepts, like diagnosis, examination, and patient are all given by definitions of collections of specific clinical terms. Knowledge acquisition is modeled as acts of defining a series of descriptive parameters,

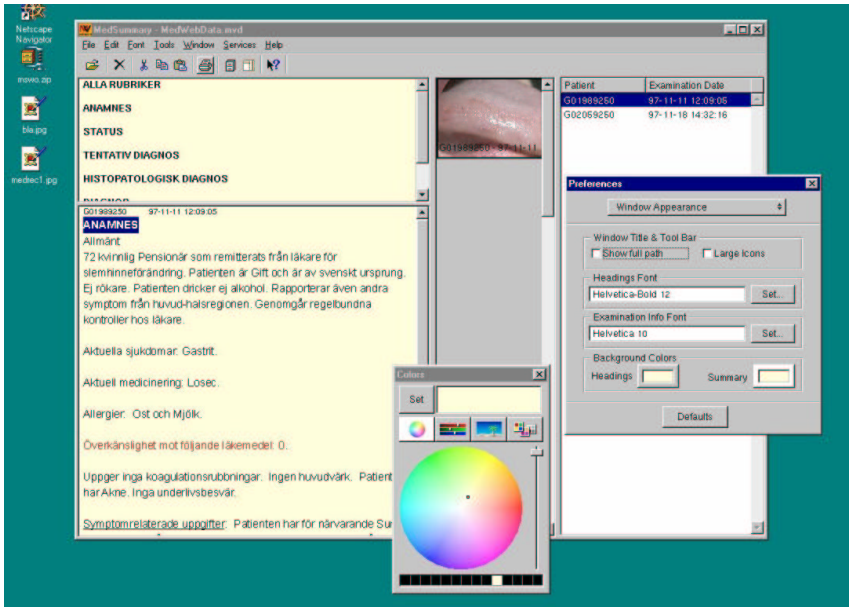

Fig. 1. MedSummary: main and preferences windows. Different texts may be generated by selecting from the headings at the top left. Clicking on a minimized image will show it full-sized in a separate window.

e.g., anamnesis (disease history), status and diagnosis, in terms of observed values. The basic data thus gathered is collected in the form of definitions which are stored into a large knowledge base. The knowledge base also contains additional definitions describing general medical knowledge.

Status. Today, the system has been in use for a couple of years at several clinics and data from more than 2000 examinations has been collected during patient visits. Apart from applications for use in the clinical setting a number of tools for exploring the database using information visualization techniques have been developed. The most used applications are

- MedRecords, which is used by clinicians to enter detailed formalized examination data during patient visits.
- MedSummary (Fig. 1), which is used to automatically generate textual and pictorial summaries of examination data using natural language generation.
- SimVis and The Cube [8], which are two analysis tools developed to enhance the clinician's ability to intelligibly analyze existing patient material and to allow for pattern recognition and statistical analysis.

Work on a definitional framework for case based reasoning is under way [7].

3 Gisela - a Framework for Definitional Programming

While GCLA is a nice realization of definitional programming it has several drawbacks which became significant when new programming techniques [6] were developed and attempts were made to use GCLA for knowledge representation and reasoning in the MedView project. Most notable were efficiency problems with certain classes of definitions and the problem of interacting well with the tools used to build GUIs.

It was therefore decided that a new definitional programming system should be developed. When we analyzed our needs we realized that what we wanted was not yet another declarative programming language based on a definitional model. Rather, we wanted to create a general framework for definitional programming which would allow us to implement definitional knowledge structures in a flexible way. Equally important was that the framework should allow us to easily build state-of-the-art desktop and web applications with embedded definitional reasoning components.

3.1 Key Properties

The analysis mentioned above led to the development of a framework with the following features important for building real applications:

- The computational model of Gisela treats definitions as abstract objects only, allowing for different kinds of implementation and behaviors.
- Gisela is designed and implemented as an object-oriented framework written in Objective-C.
- Gisela provides a complete object-oriented application programming interface (API) for building definitional components for use in applications.
- Using Gisela in various contexts, e.g,. in desktop or web applications, entails nothing more than allocating some objects, no particular interfacing techniques are needed.
- To allow for "traditional declarative programming" a second API using equational representations is provided.
- Since the computational model is given at a rather abstract level it allows for modification by subclassing the classes defined in the framework allowing for new kinds of definitions and computation methods.

Furthermore, the framework:

- Makes use of the notion of an *observer*, which sets up hooks for interactive computations.
- Gives a description of definitional programming that breaks the links to Prolog present in GCLA.
- Keeps the distinction between declarative and procedural parts of programs used in GCLA, thus separating declarative descriptions and control information.

- Allows any number of distinct definitions in programs. The GCLA system used two: the declarative (object) definition and the procedural (rule) definition.
- Allows for new definitional programming ideas while keeping many techniques developed for GCLA.

Gisela can to some extent be seen as a heir to GCLA. However, the computational model is rather different and the implementation as an extensible object-oriented framework has very little in common with the Prolog-based GCLA system.

3.2 Integration with Objective-C

Gisela was from the start designed to make it simple to build programs directly as objects, from components and classes in the framework instead of using traditional syntactic representations. Indeed, all that is required to use the Gisela framework in an Objective-C program is to create a new instance of the class `DFDMachine`, some data and method definition objects and start computing.

The general idea behind the Objective-C interface to Gisela is that each kind of entity used to build programs, variables, terms, conditions, definitions, etc., is represented by objects of a corresponding class. Thus, a constant is represented by an object of the class `DFConstant`, an equation by an object of the class `DFEquation` and so on. It follows that if we have a conceptually clear definitional model of a system it can be realized directly using object representations.

For example, the equation, `a = b.` is created by the following code segment:

```
DFConstant *a = [DFConstant constantWithName:@"a"];
DFConstant *b = [DFConstant constantWithName:@"b"];

DFEquation *eq = [DFEquation equationWithLeft:a andRight:b];
```

The basic API for using a `DFDMachine` performing actual computations is very simple consisting of the methods:

```
// Create a machine that uses the default observer.
- (id)initWithDelegate:(id)anObject;

// Set the query to evaluate.
- (void)setQuery:(DFQuery *)aQuery;

// Returns the next answer if there is one or nil otherwise.
- (DFAnswer *)nextAnswer;

// Returns an  array with all the possible answers.
- (NSArray *)findAllAnswers;
```

The framework provides a number of different definition classes for various purposes. In case these are not sufficient they can be subclassed or simply replaced by custom classes.

3.3 Computing Basics

Giving a full account of Gisela is beyond the scope of this paper. Here we present some basics and a few small examples. For a complete description see [15].

Definitions. A definition D is given by

1. two sets: the *domain* of D, written $dom(D)$, and the *co-domain* of D, written $com(D)$, where $dom(D) \subseteq com(D)$,
2. and a *definiens* operation: $D : com(D) \to \mathcal{P}(com(D))$.

Objects in $dom(D)$ are referred to as *atoms* and objects in $com(D)$ are referred to as *conditions*.

A natural presentation of a definition is that of a (possibly infinite) system of equations

$$D \begin{cases} a_0 = A_0 \\ a_1 = A_1 \\ \quad \vdots \qquad n \geq 0, \\ a_n = A_n \\ \quad \vdots \end{cases}$$

where atoms, $a_0, \ldots, a_n, \ldots \in dom(D)$, are defined in terms of a number of conditions, $A_0, \ldots, A_n, \ldots \in com(D)$, i.e., all pairs (a_i, A_i) such that $A_i \in D(a_i)$, and $a_i \in dom(D)$. Note that an equation $a = A$ is just a notation for A being a member of $D(a)$. Expressed differently, the left-hand sides in an equational presentation of a definition D are the atoms for which $D(a_i)$ is not empty.

Given a definition D the presentation as a system of equations is unique modulo the order of equations. However, given an equational presentation of a definition it is not generally possible to determine which definition the equations represent.

Intuitively, the definiens operation gives further information about its argument. For an atom a, $D(a)$ gives the conditions defining a, that is, $D(a) = \{A \mid (a = A) \in D\}$. For a condition $A \in com(D) \setminus dom(D)$, $D(A)$ gives the constituents of the condition.

Computations. Programs in Gisela consist of an arbitrary number of *data definitions* and *method definitions*. The data definitions describe the declarative content of the program, and the method definitions give the algorithms, or search strategies, used to compute solutions.

Method Definitions. All method definitions presented in this paper will be of the form

$$\begin{aligned} &\texttt{method}\ m(D_1, \ldots, D_n).\ n \geq 0 \\ &\qquad m = C_1\ \#\ G_1 \\ &\qquad\quad \vdots \\ &\qquad m = C_k\ \#\ G_k \end{aligned}$$

where m is the name of the method definition, $D_1, \ldots, D_n$, are parameters representing the actual data definitions used in computations, each C_i is a computation condition describing a number of operations to perform, and the G_is are guards restricting the applicability of equations in the method definition.

Queries. A computation is a transformation of an initial *state* definition into a final *result* definition. To compute a query, a method definition is applied to an initial state definition. We will write the initial definition as a sequence of equations. The general form of a query is

$$m(D_1, \ldots, D_n)\{e_1, \ldots, e_n\}.$$

where m is a method definition, $D_1, \ldots, D_n$ are the actual data definitions used in the computation, and each e_i is an equation.

If the computation method applied cannot be used to transform the initial state definition into an acceptable result definition the computation fails. If the computation succeeds we take the result definition and any bindings of variables in the initial state definition as the answer to the query. Note that the computation method m provides the particular data definitions describing declarative knowledge to use in the computation. Depending on the context the result of a computation can be interpreted in different ways.

Sample Programs. Programs in Gisela can be built directly from objects programmed in Objective-C based on classes in the framework or using a traditional declarative programming style. We show some simple examples using the later approach to give a flavor of the system.

Example 1. A Toy Expert System. Consider the following toy expert system adopted from [2]. The knowledge base of the system is the data definition:

```
definition diseases.

symptom(high_temp) = disease(pneumonia).
symptom(high_temp) = disease(plague).
symptom(cough) = disease(pneumonia).
symptom(cough) = disease(cold).
```

The data definition, named `diseases`, contains the connections between symptoms and diseases, but no facts. To ask the system what a possible disease might be, based on observed facts, e.g. symptoms, we form a query using a method definition and an initial state definition. For instance, assume that the patient has the symptom `high_temp`, from which diseases does this follow?

```
lra(diseases){disease(X) = symptom(high_temp)}.
```

The meaning of the query is "use the method definition `lra` instantiated with the domain knowledge in the data definition `diseases` to compute a result definition from the initial state definition `{disease(X) = symptom(high_temp)}`".

```
G3> lra(diseases){disease(X) = symptom(high_temp)}.
X = pneumonia
? ;
X = plague
```

The answer tells us that `high_temp` could be caused by `pneumonia` or `plague`. Since the result definition is of no particular interest the system has been asked to display only the computed answer substitution.

The method definition `lra` is defined as follows:

```
method lra(D).
    lra = [lra, l:D] # some l:in_dom(D).
    lra = [lra, r:D] # some r:in_dom(D) & all not(l:in_dom(D)).
    lra = [D] # all not(l:in_dom(D); r:in_dom(D)).
```

The first line states that `lra` is a method definition that takes one parameter, a data definition `D`. The remaining lines are three equations that describe the behavior implemented by `lra`.

The method `lra` attempts to replace left and right-hand sides of equations in the current state definition using the data definition `D`. If this is not possible, the third equation of `lra` will try to unify the left and right-hand side of some equation in the state definition. If no equation of `lra` can be applied, the answer to the query is `no`.

Example 2. Default Reasoning. Assume we know that an object can fly if it is a bird and if it is not a penguin. We also know that Tweety and Polly are birds as are all penguins, and finally we know that Pengo is a penguin. A data definition expressing this information is the following:

```
definition birds:gcla.

flies(X) =
   bird(X),
   (penguin(X) -> false).

bird(tweety).
bird(polly).
bird(X) = penguin(X).

penguin(pengo).
```

If we want to know which birds can fly, we pose the query

```
G3> gcla(birds){true = flies(X)}.
```

which gives the expected answers. More interesting is that we can also infer negative information, i.e., which birds cannot fly:

```
G3> gcla(birds){flies(X) = false}.
X = pengo
```

The method definition `gcla` is an attempt to emulate general GCLA behavior. More details about how the kind of negation used works can be found in [2].

Example 3. Hamming Distance. The examples above were adopted from GCLA programs. Since GCLA is essentially an extension to logic programming, the interesting part of the answer is the computed answer substitution for variables in the initial state definition. One of the objectives of Gisela is to allow for other ways of computing with definitions, where the computed result definition is the interesting part of the answer. Studying properties of definitions, such as similarity, is an example of this.

Hamming distance is a notion generally used to measure difference with respect to information content. The Hamming distance between two code-words, for example `1101` and `0110`, is the number of positions where the words differ. In this example we let each code-word be represented by a data definition. Thus, the word `1101` and the word `0110` are represented by:

```
definition w1.                  definition w2.

w(0) = 1.                       w(0) = 0.
w(1) = 1.                       w(1) = 1
w(2) = 0.                       w(2) = 1.
w(3) = 1.                       w(3) = 0.
```

To compute the Hamming distance between `w1` and `w2` we ask the query

```
G3> lr(w1,w2){w(0)=w(0), w(1)=w(1), w(2)=w(2), w(3)=w(3)}
```

which computes the result definition `{1=0, 1=1, 0=1, 1=0}`. What we have computed is, so to speak, how *similar* `w1` and `w2` are. From this similarity measure, it is easy to see that the Hamming distance is 3.

The method `lr` expands the initial state definition as far as possible by replacing atoms according to the actual data definitions used. When a state where no equation can be changed is reached the computation stops.

4 MedView and Gisela

4.1 Application Architecture

The general design approach used in applications is to use the Model-View-Controller (MVC) paradigm. MVC is a commonly used object-oriented software development methodology. When MVC is used, the *model*, that is, data and operations on data, is kept strictly separated from the *view* displayed to the user. The *controller* connects the two together and decides how to handle user actions and how data obtained from the model should be presented in the view.

Applied to the MedView and Gisela setting, the model of what an application should do is implemented using definitional programming in Gisela. The view displayed to the user can be of different kinds, desktop applications, web

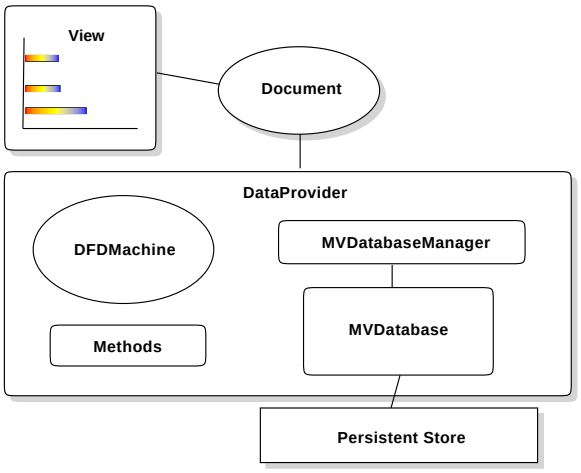

Fig. 2. An example data model.

applications etc. In between the view presented to the user and the Gisela machinery there is a controller object which manages communication between the two parts.

One advantage of this approach is, of course, that different views may be used without changing the model. The general architecture is illustrated in Fig. 2. At the center is a `Document` (controller) object, which manages an on-screen window displaying graphs and all resources needed to perform computations. All definitional computations are embedded into a model consisting of an object of the `DataProvider` class. An object of this class creates a `DFDMachine` to perform computations and feeds it with data from an `MVDatabase`. The `MVDatabase` in turn is provided by an `MVDatabaseManager`, which is responsible for things like loading databases and sharing them among objects. Gisela also provides API to load definition objects at run time from text-files. Consequently, any Gisela program developed using equational presentations can smoothly be integrated into an Objective-C application.

4.2 Knowledge Base Structure

The MedView knowledge base consists of a large number of definitions stored in a format that can be read by the Gisela framework. The general structure of the knowledge base is pictured in Fig. 3. It consists of the following:

- A collection of examination records, where each examination is represented by a data definition.
- Additional data definitions describing different kinds of general knowledge.
- Procedural knowledge represented by definitions in terms of method definitions.

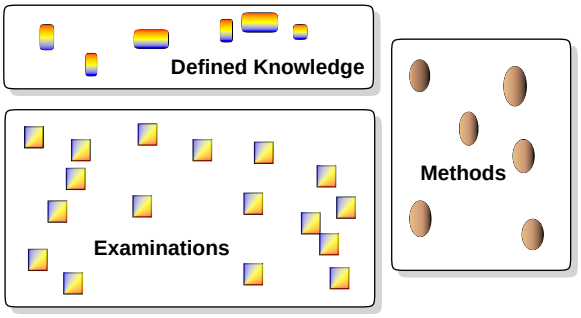

Fig. 3. Schematic view of the MedView knowledge base. The knowledge base consists of a collection of examination records on top of which extra knowledge may be added. To perform computations, methods, shown to the right, are needed.

Note the distinction between declarative and procedural knowledge. In addition, the knowledge base contains a large number of digitized images taken at examinations. Each image is associated with a particular examination and can be retrieved by searching the collection of examination records.

Representing Basic Data. Assembling information at examinations is modeled as defining a series of descriptive parameters, such as disease history (anamnesis), status, diagnosis, and so on. All examination definitions share a common structure, which so to speak defines the basic concept "examination". A small part of this structure is:

$$E \begin{cases} examination = anamnesis \\ examination = status \\ anamnesis = common \\ status = direct \\ common = drug \\ common = smoke \\ direct = mucos \\ direct = palpation \\ mucos = mucos\_site \\ mucos = mucos\_col \\ palpation = palp\_site \end{cases} \tag{1}$$

As can be seen from the example, the general structure is hierarchical. An examination consists of *anamnesis*, *status*, etc. Each of these in turn consist of a number of parts, which consist of a number of parts, and so on until we reach the actual *attributes* for which values are collected at an examination. The attributes in (1) are, *drug*, *smoke*, *mucos_site*, *mucos_col*, and *palp_site*. It is important to note that not all attributes must be given values. A missing value simply indicates that we know nothing about it. The structure may be changed as long

as it is *extended*, since then old records will remain valid, it is simply that they may have a larger number of attributes without values.

It is natural to view an examination record as being the sum of two definitions. One definition, E, describing the concept examination, and one definition, R, providing data collected at a particular patient visit. Thus, a complete examination is given by $E + R$.

For instance, a set of equations that together with (1) define a particular examination could be

$$R \begin{cases} drug = losec \\ drug = dermovat \\ smoke = no \\ mucos\_site = l122 \\ mucos\_col = white \end{cases}$$

4.3 Additional Knowledge Structures.

On top of the basic collection of examination records additional definitions may be created to represent different kinds of knowledge. We show two examples.

Value Classes. As the number of examinations in the knowledge base grows, it becomes increasingly important to group related values into classes in a hierarchical manner. For example, diseases such as Herpes labialis, Herpetic gingivostomatis, Shingles etc., can be classified into viral diseases. Such classifications facilitate the detection of interesting patterns in the data. Value classes are given as definitions and can be stored in the knowledge base for future use. Examples of existing simple value classes are a division between smokers and non-smokers and between patients with oral lichen planus and patients that do not have oral lichen planus.

A Gisela definition that classifies smoking habits into three groups is:

```
definition smoke_3:constant.

'1 cigaretter utan filter/dag' = '< 10 cigarettes/day'.
'4 cigaretter utan filter/dag' = '< 10 cigarettes/day'.
'10 filtercigaretter/dag' = '> 10 cigarettes/day'.
'10-15 filtercigaretter/dag' = '> 10 cigarettes/day'.
'40 filtercigaretter/dag' = '> 10 cigarettes/day'.
'Nej' = 'non smoking'.
```

Of course, as the knowledge base grows so will the number of equations in the definition `smoke_3`. To further categorize smoking habits into smokers and non-smokers another definition can be used:

```
definition smoke_2:constant.

'< 10 cigarettes/day' = smoking.
'> 10 cigarettes/day' = smoking.
```

Note that in the definition `smoke_2` it is assumed that smoking habits have already been grouped using `smoke_3`. A complete value class definition is derived by adding together `smoke_2` and `smoke_3`, that is, conceptually $smoke = smoke\_2 + smoke\_3$.

To examine the use of value classes consider the query

$$m(D_1, \ldots, D_n)\{V = A\}$$

where m is a computation method using the definitions $D_1, \ldots, D_n$, V is the value we are looking for and A some kind of attribute. We will say that a record fulfills the demands of the query if the single equation in the initial state definition can be reduced to identity. For instance a query could be

```
srfi(e1297,smoke_3){'<10 cigarettes/day' = 'Smoke'}.
```

which succeeds if the examination represented by definition `e1297` has a value for the attribute `'Smoke'` indicating that the patient smokes less than 10 cigarettes a day.

A possible method definition encoding the necessary procedural knowledge is:

```
method srfi(Exam,Filter).
    srfi = [] # some identity.
    srfi = [srfi, r:Exam] # some r:in_dom(Exam) &
                            all not(identity).
    srfi = [srfi, r:Filter] # some r:in_dom(Filter) &
                              all not(identity) &
                              all not(r:in_dom(Exam)).
```

The meaning of the equations in `srfi` is: (i) if there is an equation with identical left- and right-hand sides in the current state definition, the computation is finished, (ii) if some attribute can be reduced using `Exam`, reduce it and continue, and (iii) if a value can be grouped using `Filter`, do that and continue.

Computationally, in the query above, `'Smoke'` is first replaced with the value in the definition `e1297`, and then this value is replaced by the appropriate group as given by the definition `smoke_3`.

If we add the definitions `smoke_3` and `smoke` together (an operation supported by the Gisela framework) we could ask the query

```
srfi(e1297,smoke_3+smoke_2){smoking = 'Smoke'}.
```

to check whether a person smokes or not.

Value Corrections. In the MedView project new values for attributes may be freely added by any user. This is necessary since we cannot anticipate all possible values. Also, it is not possible to wait for approval when a new value is encountered, since data is entered during examinations. There are of course at least two major problems with this practice: (i) letting all users add new values

might lead to confusion and a less harmonized terminology within the network of users involved, (ii) misspelled words may be introduced into the lists of valid values.

One solution is to monitor the values in the knowledge base regularly and add extra definitions that can be used to find replacements for incorrect values. These definitions can then be used by applications to ensure that a harmonized terminology is used. It is natural to use one definition with corrections for each attribute in the general examination structure that has incorrect values. Which values are correct and which are not is decided by the network of clinicians working with MedView. Solving the problem by making changes in the examination definitions directly is not a viable solution for several reasons, one being the general rule stating that medical record information may not be changed.

As an example, when the values used for the attribute `'Chld-dis'` (Child Disease) were inspected, it was found that both "Mässlingen" and "Mässling" were used to denote the same disease (Measles). It was also found that in a number of examinations the disease "Röda Hund" (Rubella) was misspelled "Ruda Hund".

A Gisela definition that describes how to correct values for the attribute `'Chld-dis'` is the following:

```
definition 'Chld-dis'.

'Mässling' = 'Mässlingen'.
'Ruda hund' = 'Röda Hund'.
```

It was decided that "Mässlingen" was preferred over "Mässling". Therefore, the value `'Mässling'` is defined to be equal to the correct value `'Mässlingen'`. Note that only values that are regarded as incorrect are defined in this definition. For simplicity, the name of the definition is the same as the name of the attribute it gives corrections for. From a computational point of view value corrections are essentially identical to value classes.

5 Discussion

The value of Gisela and its use in the MedView project depends on two key issues: (i) To what extent the use of a uniform declarative model facilitates the development of a system such as MedView, and (ii) How well the Gisela system is suited for the task at hand. These issues are discussed briefly here.

5.1 Definitions and MedView

The choice to use definitions as the model underlying knowledge representation and reasoning within MedView was taken in a very early phase of the project based on the expertise of the computer scientists involved. However, the model has proven easy for clinical users to understand and modify. Today, development and maintenance of the general structure of examinations, value lists, value and

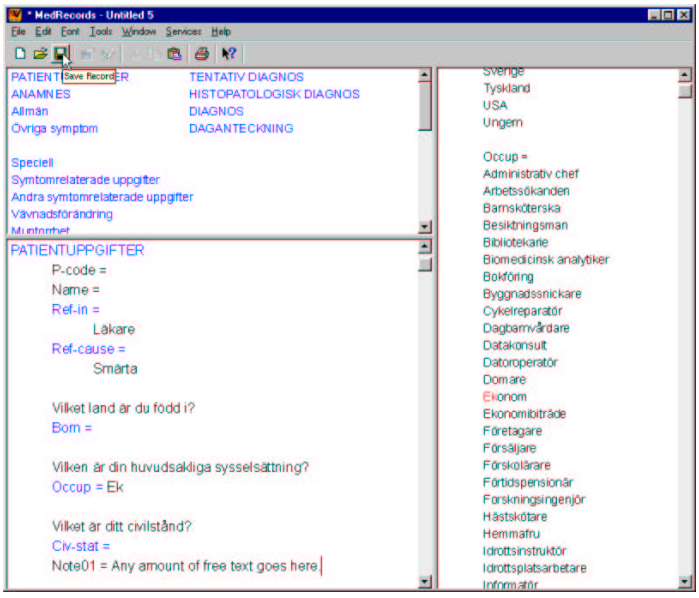

Fig. 4. MedRecords. At the top left is a navigation area which is used to navigate into the appropriate part of the input view (incomplete definition) at the bottom. To the right is a list of values linked to the attributes in the input view.

correction definitions etc. is managed by the involved clinicians themselves. An example of this is the MedRecords application shown in Fig.4 used to enter data at examinations. The idea behind this application is to display an incomplete definition (examination record) and provide means to facilitate completing it. The contents of the three views is developed by the users themselves. It is also worth noting that although MedRecords does not make use of Gisela it is definitely colored by the declarative model used.

Clinicians using MedView typically report increased competence resulting from the stringent procedures set-up by using a formalized nomenclature and standardized examination protocols. Furthermore, clinicians rarely have to dictate medical record texts since these can be generated from examination records.

5.2 The Gisela Framework

A major advantage of using declarative programming is that the close gap between the knowledge model and programming model greatly simplifies implementing the reasoning part of the system. While a large number of examinations have been collected in MedView, the work on additional knowledge structures is still in an early phase. In part, this depends on that, prior to the development of the Gisela framework, each kind of new definitional computation to be performed required the development of new specialized procedures for computing

with definitions. Accordingly, trying out new ideas in practice was a cumbersome process that required a lot of work. With the Gisela framework different kinds of definitions and computation methods can be expressed easily, something we hope will speed-up the process of trying out new ideas on how to explore the MedView knowledge base.

As the database is approaching a size where it might be meaningful to apply data mining techniques to search for patterns it will become increasingly important to have a tool that is tailored for definitional computing. It will also make it easier to implement, for instance, an intelligent agent in MedRecords helping the user during patient visits.

Of course, most declarative languages have foreign language interfaces. However, there are two things that set Gisela apart most of these: (i) Gisela does not attempt to be a general-purpose programming language, rather it is a system for realizing a certain set of definitional models, (ii) Gisela is a framework with a rather loose definition, specifically aimed at allowing experiments and modifications within the general model set up by the computation model. The aim of declarative systems such as Prolog, Haskell, Mercury [13], and Curry [11], is to provide full-fledged programming languages suitable as alternatives to the commonly used imperative and object-oriented ones. Being general-purpose languages, they also provide libraries to build GUIs [4, 10]. An alternative path is taken in [5] which implements Prolog in Java in a manner enabling a tight integration between the two languages.

So far, our experiences from using Gisela are positive: It provides seamless integration with state-of-the-art tools for building desktop and web applications, handles the current database without any problems, and can easily be modified or extended when functionality not present in the framework is needed. If desired the framework can also be used to use declarative programming at different levels. For instance, it is possible to use only data definitions describing a domain and implement the procedural behavior without using Gisela.

5.3 Conclusion

We have described how the declarative programming tool Gisela is used for knowledge representation within the MedView project. The approach taken has been to build a declarative programming tool which can be integrated with ease into a modern object-oriented programming environment. Currently, we have no plans to extend Gisela to handle sophisticated interaction with the user. Instead, we advocate an approach with a definitional model programmed in Gisela and an interface part programmed using other, more suitable, tools. In our experience this is the most practical approach, at least for the time being.

So far, more than 2000 examinations and some 2500 images have been collected into the knowledge base, which in the area of oral medicine is a significant contribution. For the future, both Gisela and MedView are being ported to Java and we are looking into different ways to model and use the knowledge base accumulated during the years.

References

1. Y. Ali, G. Falkman, L. Hallnäs, M. Jontell, N. Nazari, and O. Torgersson. Medview: Design and adoption of an interactive system for oral medicine. In A. Hasman, B. Blobel, J. Dudeck, R. Engelbrecht, G. Gell, and H.-U. Prokosch, editors, *Medical Infobahn for Europe: Proceedings of MIE2000 and GMDS2000*. IOS Press, 2000.
2. M. Aronsson. Methodology and programming techniques in GCLA II. In *Extensions of logic programming, second international workshop, ELP'91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.
3. M. Aronsson, L.-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin. The programming language GCLA: A definitional approach to logic programming. *New Generation Computing*, 7(4):381–404, 1990.
4. M. Carlsson and T. Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, 1993.
5. E. Denti, A. Omicini, and A. Ricci. tuProlog: a light-weight prolog for internet applications and infrastructures. In *Proc. of the Third International Workshop on Practical Aspects of Declarative Languages (PADL'01)*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer-Verlag, 2001.
6. G. Falkman. Program separation and definitional higher order programming. *Computer Languages*, 23(2–4):179–206, 1997.
7. G. Falkman. Similarity measures for structured representations: a definitional approach. In E. Blanzieri and L. Portinale, editors, *EWCBR-2K, Advances in Case-Based Reasoning*, Lecture Notes in Artificial Intelligence, pages 380–392. Springer–Verlag, 2000.
8. G. Falkman. Information visualization in clinical odontology: multidimensional analysis and interactive data exploration. *Artificial Intelligence in Medicine*, 22(2):133–158, 2001.
9. L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.
10. M. Hanus. A functional logic programming approach to graphical user interfaces. In *Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, volume 1753 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, 2000.
11. M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
12. P. Kreuger. GCLA II: A definitional approach to control. In *Extensions of logic programming, second international workshop, ELP91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.
13. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
14. O. Torgersson. A definitional approach to functional logic programming. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Extensions of Logic Programming 5th International Workshop, ELP'96*, number 1050 in Lecture Notes in Artificial Intelligence, pages 273–287. Springer-Verlag, 1996.
15. O. Torgersson. *On GCLA, Gisela, and MedView: Studies in Declarative Programming with Application to Clinical Medicine*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2000.