IMPERATIVE HISTORY: TWO-DIMENSIONAL EXECUTABLE TEMPORAL LOGIC

MARCELO FINGER
Departamento de Ciencia da Computacao
Instituto de Matematica e Estatistica
Universidade de Sao Paulo

AND

MARK REYNOLDS
Department of Computer Science
King's College, London

1 Introduction

In this paper we combine two interesting and useful recently proposed ideas within applied temporal logic which were both initially developed by Dov Gabbay (amongst others). We coin the term "Imperative History" for the two-dimensional executable temporal logic which results from combining the Imperative Future idea of an executable temporal logic (proposed in [10] and described more fully in [3]) with the idea of using a two-dimensional temporal logic to describe the evolution of temporal databases (an idea proposed in [6] but closely related to the work in [4]). We demonstrate that this combination leads to a powerful declarative approach to handling time in databases.

Temporal Logic has become one of the most important formalisms for describing, specifying, controlling and reasoning about systems which exhibit some kind of on-going interaction with their environment. The formal language with its proof-theory, decision algorithms and associated methods of practical application has found many uses in dealing with programs, complex reactive systems, databases and artificial intelligent systems: the interested reader is referred to [9] for a fuller description of these applications. In this paper we extend two different applications.

In [10] it was suggested that the formal temporal language for describing the development of a reactive system could be used, in a restricted form, to actually write the programs which control the behaviour of the sys-

tem. Thus we use temporal logic as a declarative programming language: the logic becomes executable. All the well-known advantages of declarative programming languages apply: they are quick to write, easy to understand and anyone interested in formal verification has a head start.

In the executable temporal logic of [10], the simple restricted format for the formulas of the temporal language which become program rules is summarized as *Past* implies *Future*. The procedural effect of such a rule is that some condition on the observed past behaviour of the system (and/or its environment) controls whether the system brings about some future situation. Thus this idea is rendered as *Declarative Past* implies *Imperative Future*. There is an ever increasing body of useful work developing from this proposal and related work. The interested reader can find descriptions of first-order versions, efficient implementations and generalizations to concurrency amongst other recent developments in [3].

Another very important use of temporal logic is in dealing with databases which make use of time. We call these *temporal databases*. Time can be relevant to a database in one or both of two different ways. Each change to the contents of the database will be made at some time: we refer to this as the *transaction* time of the database update. Databases often also store information about the time of events: we refer to the actual time of occurrence of an event as its *valid* time. Depending on which of these uses is made of time or on whether both approaches have a role to play, we can identify several different types of temporal databases but what is common to all, as with all systems which change over time, is that describing or reasoning about their evolution is very conveniently done with temporal logic.

With both the forms of temporal information involved, it was thus suggested in [6], that describing the evolution of a temporal database is best done with two-dimensional temporal logic. This is because, for example, at a certain transaction time today, say, we might realize that our database has not been kept up to date and we may add some data about an event which occurred (at a valid time) last week. Thus a one-dimensional model which represents this-morning's view of the history of the recorded world, is changed, by the afternoon, into a new one-dimensional model by having the state of its view about last week altered. A series of one-dimensional models arranged from one day to the next is clearly a structure for a two-dimensional temporal logic. Other applications of two-dimensional temporal logic exist— for example in dealing with intervals of time [1]— but the logic is generally quite difficult to reason with (see [22]). However, it has recently been shown ([4]) that the kind of logic needed for database applications is much more amenable.

Managing databases is not just about collecting facts. There are many uses for more general rules. For example, we often need integrity con-

straints, derived properties, conditional updates, side-effects and systematic corrections. All such rules must be expressed in some sort of database-control/programming language.

In this paper we suggest using a two-dimensional executable temporal logic as a declarative language for expressing rules for temporal database management. The most common form for these rules will be a formula which expresses a condition on the one-dimensional historical model at a certain time controlling a condition on the new one-dimensional historical model which should hold after the next transaction. This may necessitate an update to recorded history (about some valid times in the past, present or future). We thus call this executable temporal logic "Imperative History".

The paper is structured as follows. In the next section, we define propositional and predicate one-dimensional temporal logics: their languages form the basis of existing executable temporal logics and our two-dimensional temporal logic. Also in this section, we describe the existing (one-dimensional) executable temporal logic METATEM and its variations. In section 3, we describe two-dimensional logic as it is applied to temporal databases. We also briefly describe the idea of temporal databases and their various types. In section 4, we introduce the idea of an executable two-dimensional logics and describe how it could be used in database management. In section 5, we provide a simple example of the idea in action in the intensive care ward of a hospital: this example develops, some previous applications of executable temporal logic. In section 6, we give a possible extension of the technique to database triggers before summarizing our work.

2 Executable Temporal Logic

2.1 Temporal Logic

We are going to be concerned with the behaviour of processes over time. Two very useful formal languages for describing such behaviour are the propositional temporal logic **PTL** and the first-order temporal logic **FTL** based on the temporal connectives until U and since S introduced by Kamp in [12]. The simpler propositional language allows us to express less and so is easier to deal with.

A crucial point in the executable temporal logic paradigm is that the same languages are used to specify the desired behaviour of a program and to actually write the program to satisfy the specification. In fact, in the ideal case, the specification and the program are the same thing.

In any case, amongst many other advantages, using the same language for specification and implementation gives us a head start in proving correctness of programs.

2.2 Temporal Structures

The languages **FTL** and **PTL** are used to describe the behaviour of processes over time. In this paper, we will take the underlying flow of time to be either the natural numbers –equivalently some sequence $s_0, s_1, s_2, ...$ of states – or the integers. In general, such temporal languages can describe changes over any linear order $(T, <)$ of time points.

In the propositional case the *state* at each time is just given by the truth values of a set $\mathcal{L}_P$ of atomic propositions or atoms. The behaviour we are describing is just the way the various atoms become true or false over time. To formalize this we use a map $\pi_P : T \times \mathcal{L}_P \to \{\top, \bot\}$ where $\pi_P(t, q) = \top$ iff the atom q is true at time t.

In first-order temporal structures the state at each time is a whole first-order structure with a domain of objects on which are interpreted constant symbols, function symbols and predicate symbols. Without any restrictions such situations would be too messy to describe formally so we make some assumptions. As described in [18] there are many sets of simplifying assumptions which can be made but the ones we make here are comfortable to work with and, at the same time, so general that other approaches can be easily coded in.

For a start we assume that each state is a first order structure in the same language. So suppose that $\mathcal{L}_P$ is a set of predicate symbols and $\mathcal{L}_F$ is a set of function symbols. We divide up $\mathcal{L}_P$ into a set $\mathcal{L}_P^n$ for each $n \geq 0$ being the n-ary predicate symbols. We also divide up $\mathcal{L}_F$ into a set $\mathcal{L}_F^n$ for each $n \geq 0$ being the n-ary predicate symbols. The 0-ary function symbols are just constants.

We assume a constant domain $\mathcal{D}$ of objects but, over time, the extensions of the predicates change. To formalize this we use a map $\pi_P = \pi_P^0 : T \times \mathcal{L}_P^0 \to \{\top, \bot\}$ and a map $\pi_P^n : T \times \mathcal{L}_P^n \to \mathcal{D}^n$ for each $n = 1, 2, ...$ The interpretations of the functions are constant: we use maps $\pi_F^n : \mathcal{L}_F^n \to (\mathcal{D}^n \to \mathcal{D})$.

In many of the definitions below we can include the propositional case as a special case of the first-order one by equating $\mathcal{L}_P$ with $\mathcal{L}_P^0$ and π_P with π_P^0.

2.3 Syntax

As well as $\mathcal{L}_P^n$ and $\mathcal{L}_F$, we also use a countable set $\mathcal{L}_V$ of variable symbols. The terms of **FTL** are built in the usual way from $\mathcal{L}_F$ and $\mathcal{L}_V$.

The set of formulas of **FTL** is defined by:

- if $t_1, ..., t_n$ are terms and p is an n-ary predicate symbol then $p(t_1, ..., t_n)$ is a formula,

– if α and β are formulas then so are $\top$, $\neg\alpha$, $\alpha \wedge \beta$, $\forall x \alpha$, $\mathcal{U}(\alpha, \beta)$ and $\mathcal{S}(\alpha, \beta)$.

We have the usual idea of free and bound variable symbols in a formula and so the usual idea of a sentence – i.e. a formula with no free variables. The class of formulas which do not have any variable symbols or constants form the well-formed formulas of the propositional language **PTL**. In **PTL** we only use 0-ary predicate symbols which are just propositions.

A formula of the form $p(\overline{u})$ is called a *positive literal*. A formula of the form $\neg p(\overline{u})$ is called a *negative literal*. A *literal* is either a positive one or a negative one. A literal is *ground* if it is also a sentence.

2.4 Semantics

A variable assignment is a mapping from $\mathcal{L}_V$ into $\mathcal{D}$. Given such a variable assignment V we assume it extends to all terms by recursively defining $V(f(u_1, ..., u_n)) = \pi^n_F(f)(V(u_1), ..., V(u_n))$ for any $f \in \mathcal{L}^n_F$.

For a temporal structure $\mathcal{M} = (T, <, \mathcal{D}, \{\pi^n_P\}_{n \geq 0}, \{\pi^n_F\}_{n \geq 0})$ a time point $t \in T$, a formula φ, and a variable assignment V, we define whether (or not resp.) $\varphi(\overline{d})$ under V is true at t in $\mathcal{M}$, written $\mathcal{M}, t, V \models \varphi$ (or $\mathcal{M}, t, V \not\models \varphi$ resp.) by induction on the construction of φ.

– $\mathcal{M}, t, V \models \top$.
– $\mathcal{M}, t, V \models q$ for a proposition q iff $\pi^0_P(t, q) = \top$.
– $\mathcal{M}, t, V \models p(\overline{u})$ for an n-ary predicate p and n-tuple $\overline{u}$ of terms iff $(V(u_1), ..., V(u_n)) \in \pi^n_P(t, p)$.
– $\mathcal{M}, t, V \models \neg\chi$ iff $\mathcal{M}, t, V \not\models \chi$.
– $\mathcal{M}, t, V \models \chi \wedge \psi$ iff $\mathcal{M}, t, V \models \chi$ and $\mathcal{M}, t, V \models \psi$.
– $\mathcal{M}, t, V \models \mathcal{U}(\psi, \chi)$ iff there is $s > t$ in T such that $\mathcal{M}, s, V \models \psi$ and for all $r \in T$ such that $t < r < s$, $\mathcal{M}, r, V \models \chi$.
– $\mathcal{M}, t, V \models \mathcal{S}(\psi, \chi)$ iff there is $s < t$ in T such that $\mathcal{M}, s, V \models \psi$ and for all $r \in T$ such that $s < r < t$, $\mathcal{M}, r, V \models \chi$.
– $\mathcal{M}, t, V \models \forall x \chi$ for $x \in \mathcal{L}_V$ iff for all $d \in \mathcal{D}$ $\mathcal{M}, t, W \models \chi$ where W is the variable assignment given by

$$W(y) = \begin{cases} V(y) & y \neq x \\ d & y = x. \end{cases}$$

It is easy to prove that the truth of a formula at a point in a structure does not depend on assignments to variables which do not appear free in it. So we can write $\mathcal{M}, t, v \models \varphi$ where v is a partial assignment provided its domain does include the free variables of φ. When σ is a sentence – or a **PTL** formula – we also write $\mathcal{M}, t \models \sigma$ iff $\mathcal{M}, t, \varnothing \models \sigma$ where $\varnothing$ is the empty map.

2.5 Models

Say that temporal structure $\mathcal{M}$ is a *model* of a sentence σ iff $\mathcal{M}, 0 \models \sigma$. A sentence is *satisfiable* iff it has such a model. A sentence σ is *valid* iff $\mathcal{M}, t \models \sigma$ for all structures $\mathcal{M}$ and for all time points t in $\mathcal{M}$.

2.6 Abbreviations

We read $\mathcal{U}(\psi, \xi)$ as "ξ until ψ" and similarly for since. Note that our $\mathcal{U}$ is *strict* in the sense that $\mathcal{U}(q, p)$ being true says nothing about what is true now. In some presentations of temporal logic, until is defined to be non-strict. We can introduce an abbreviation $\mathcal{U}^+$ for non-strict until: $\mathcal{U}^+(\psi, \xi)$ iff $\psi \vee (\xi \wedge (\mathcal{U}(\psi, \xi)))$.

As well as the classical abbreviations $\bot$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\exists$ we also have many temporal ones. The only ones that we need in this paper are:

$\bigcirc\varphi$	"φ is true in the next state"	$\mathcal{U}(\varphi, \bot)$
$\circledcirc\varphi$	"there was a last state and φ was true in this state"	$\mathcal{S}(\varphi, \bot)$
$\Diamond\varphi$	"φ will be true in some future state"	$\mathcal{U}(\varphi, \top)$
$\Box\varphi$	"φ will be true in all future states"	$\neg\Diamond\neg\varphi$
$\blacklozenge\,\varphi$	"φ was true in the past"	$\mathcal{S}(\varphi, \top)$
$\blacksquare$	"φ has always been true in the past"	$\neg\blacklozenge(\neg\varphi)$
start	"it is now the start of time"	$\neg(\mathcal{S}(\top, \top))$

2.7 Separation

As one would expect from the declarative past /imperative future motivation, one distinction which plays an important role in METATEM is that between formulas which refer to the past and those which refer to the future. Let us make this precise.

A formula φ is a *not necessarily strict future time formula* iff it is built without $\mathcal{S}$. The class of *strict future time* formulas include only

- $\mathcal{U}(\chi, \psi)$ where ψ and χ are both not necessarily strict future time formulas and
- $\neg\varphi$, $\varphi \wedge \psi$ and $\forall x \varphi$ where φ and ψ are both strict future time formulas.

Dually we have strict and not necessarily strict past time formulas.

It is clear that a strict past time formula only depends on the past for its truth. This classification of formulas is the basis for Gabbay's separation property and separation theorem which is itself useful for establishing the expressive power of the METATEM language. See [9] for details which also include a discussion of the proof-theory of the temporal logics mentioned above.

2.8 Explicit Time

In using executable temporal logic it is often useful to be able to refer explicitly to the time of an event as measured by some clock or calendar. This is especially so when we come to use the logic to reason about temporal databases. To support this feature we will suppose that our logic includes a special 1-ary predicate `time` which at any time t is only true of some syntactic representation of t. That is, there are enough constants and function symbols in the language to allow us to write (the name of) time t and all temporal structures $\mathcal{M} = (T, <, \mathcal{D}, \{\pi_P^n\}_{n\geq 0}, \{\pi_F^n\}_{n\geq 0})$ mentioned are supposed to have the property that $\pi_P^1(t, \texttt{time}) = \{t\}$ for each $t \in T$.

Note that with this assumption on our structures, the temporal language **FTL** can easily be shown to be as expressive as a two-sorted first-order language.

2.9 METATEM Programming Language

METATEM is really a paradigm for programming languages rather than one particular language. The bases are three:

- programs should be expressed in a temporal language;
- programs should be able to be read declaratively;
- the operation of the program should be interpretative with individual program clauses operating according to the "declarative past implies imperative future" idea.

Most versions of METATEM use the temporal languages **PTL** and **FTL** with until and since.

The basic idea of declarative languages is that a program should be able to be read as a specification of a problem in some formal language and that running the program should solve that problem. Thus we will see that a METATEM program can easily be read as a temporal sentence and that running the program *should* produce a model of that sentence.

The task of the METATEM program is to build a model satisfying the declared specification. This can sometimes be done by a machine following some arcane, highly complex procedure which eventually emerges with the description of the model (see [16]). That would *not* be the METATEM approach. Because we are describing a programming language, transparency of control is crucial. It should be easy to follow and predict the program's behaviour and the contribution of the individual clauses must be straight forward.

Fortunately, these various disparate aims can be very nicely satisfied by the intuitively appealing "declarative past implies imperative future" idea of [10]. The METATEM program *rule* is of the form $P \Rightarrow F$ where P

is a strict past-time formula and F is a not necessarily strict future-time formula. The idea is that on the basis of the declarative truth of the past time P the program should go on to "do" F.

In the case of a closed system, a METATEM program is a list $\{P_i \rightarrow F_i \mid i = 1, ..., n\}$ of such rules and, at least in the propositional case, it represents the **PTL** formula

$$\Box \bigwedge_{i=1}^{n} (P_i \rightarrow F_i).$$

The program is read declaratively as a specification: the execution mechanism should deliver a model of this formula. To do so it will indicate which propositions are true and which are false at time 0, then at time 1, then at time 2, e.t.c. It does this by going through the whole list of rules $P_i \rightarrow F_i$ at each successive stage and make sure that F_i gets made true whenever P_i is. This is called *forward chaining*. For details of the way it does this see [2].

In the first-order case there are several slightly different versions of METATEM. The simplest involves allows only clauses in the forms:

$$\textbf{start} \Rightarrow p(\overline{c})$$

$$\textbf{start} \Rightarrow \Diamond q(\overline{c})$$

$$\forall \overline{X}.[\, \textcircled{\scriptsize o} \bigwedge_{i=1}^{h} k_i(\overline{X}) \Rightarrow p(\overline{X})]$$

$$\forall \overline{X}.[\, \textcircled{\scriptsize o} \bigwedge_{i=1}^{h} k_i(\overline{X}) \Rightarrow \Diamond q(\overline{X})]$$

where each k_i is a literal, p and q are predicates and $\overline{c}$ is a tuple of constants. These constraints enable us to implement the program in a direct way.

3 Temporal Updates

We define a *temporal database* as a *finite* temporal structure, i.e. a temporal structure $\mathcal{M} = (T, <, \mathcal{D}, \{\pi_P^n\}_{n \geq 0}, \{\pi_F^n\}_{n \geq 0})$ obeying the following constraints:

- The set of predicate symbols is finite.
- The interpretation of each predicate is finite, i.e. for every predicate symbol p there are only finitely many tuples $\langle a_1, \ldots, a_n \rangle$ for which there exists a $t \in T$ such that

$$\pi_P^n(t, p(a_1, \ldots, a_n)) = \top$$

- It is usual for databases that $\mathcal{M}$ be a *Herbrand model*, i.e. the domain $\mathcal{D}$ is identical to the set of constant symbols and every ground term is interpreted into itself.
- We further assume that the set of points where an atomic formula $p(a_1, \ldots, a_n)$ is true must be representable by a *temporal element*, i.e. a finite union of intervals over T. For example, if $T = \mathbb{N}$, then $[0, 10] \cup [20, 30]$ is a temporal element, but the set of all even numbers is not. [1]

These conditions guarantee that temporal databases can be finitely represented as a set of *labelled formulas*, $i : p(a_1, \ldots, a_n)$, where $p(a_1, \ldots, a_n)$ is an atomic formula and i is a temporal element over T; in this context, a labelled formula $i : p(a_1, \ldots, a_n)$ represents a partial temporal structure.

In traditional databases, the update of data means the replacement of the current value of the data by a new one. In temporal databases one is presented with the extra possibility of changing the past, the present and the future. In a nutshell, a temporal update is a "change in history". Note the double reference to time in such an expression: *change* relates to the temporal evolution of data, while *history* refers to the temporal record. These two notions of time are independent and coexistent. In analysing updates in temporal databases we have to be able to cope simultaneously with those two notions of time. For that, we present next a *two-dimensional temporal logic*, a formalism that will allow for the simultaneous handling of two references of time. We will stick to the propositional case, for the updates we are concerned with are only atomic updates that may be modelled as propositional atoms.

3.1 Propositional Two-dimensional Temporal Logic

There are several modal and temporal logic systems in the literature which are called *two-dimensional*; all of them provide some sort of double reference to an underlying modal or temporal structure. More systematically, two-dimensional systems have been studied as the result of combining two one-dimensional logic systems [4, 5]. In [5] two criteria were presented to classify a logical system as two-dimensional:

- *The connective approach:* a temporal logic system is two-dimensional if it contains two sets of connectives, each set referring to a distinct flow of time.

[1] This condition basically tells that there is a limit to what temporal data can be represented. The condition itself can be relaxed if data expressivity is enhanced, but some time-stamps will always remain unrepresentable. Eg. if deductive rules are added to the database, periodic sets, like the even numbers, become representable; but since there are uncountably many subsets of $\mathbb{N}$, it will never be possible to represent all of them.

– *The semantic approach:* a temporal logic system is two-dimensional if the truth value of a formulas is evaluated with respect to two time points.

The two criteria are independent and there are examples of systems satisfying each criterion alone, or both. For the purposes of this work, the two-dimensional temporal logic satisfies both criteria, and is thus a *broadly two-dimensional logic*. Both flows of time are assumed to be discrete ($\mathbb{Z}$). In databases it is usual to use $\mathbb{Z}$ instead of $\mathbb{N}$ as the underlying flow of time.

So let $\mathcal{L}$ be a countable set of propositional atoms. Besides the boolean connectives, we consider two sets of temporal operators. The *horizontal operators* are the usual "since" ($\mathcal{S}$) and "until" ($\mathcal{U}$) two place operators, together with all the usual derived operators; the horizontal dimension will be used to represent *valid time* temporal information. The *vertical dimension* is assumed to be a $\mathbb{Z}$-like flow and the operators over such dimensions are the two-place operators "since vertical" ($\overline{\mathcal{S}}$) and the "until vertical" ($\overline{\mathcal{U}}$); in general, we use barred symbols when they refer to the vertical dimension. The vertical dimension will be used to represent *transaction time* information. Two-dimensional formulas are inductively defined as:

– every propositional atom is a two-dimensional formula;
– if A and B are two-dimensional formulas, so are $\neg A$ and $A \wedge B$, $\mathcal{S}(A,B)$ and $\mathcal{U}(A,B)$, $\overline{\mathcal{S}}(A,B)$ and $\overline{\mathcal{U}}(A,B)$.

On the semantic side, we consider two flows of time: the horizontal one $(T,<)$ and the vertical one $(\overline{T},\overline{<})$. Two-dimensional formulas are evaluated with respect to two dimensions, typically a time point $t \in T$ and a time point $\bar{t} \in \overline{T}$, so that a *two dimensional plane model* is a structure based on two flows of time $\mathcal{M} = (T, <, \overline{T}, \overline{<}, \pi)$. The *two-dimensional assignment* π maps every triple $(t, \bar{t}, p)$ into $\{\top, \bot\}$. The model structure can be seen as a two-dimensional plane, where every point is identified by a pair of coordinates, one for each flow of time (there are other, non-standard models of two-dimensional logics which are not planar; see [5]).

The fact that a formula A is true in the two-dimensional plane model $\mathcal{M}$ at point $(t, \bar{t})$ is represented by $\mathcal{M}, t, \bar{t} \models A$ and is defined inductively as:

$\mathcal{M}, t, \bar{t} \models p$ iff $\pi(t, \bar{t}, p) = \top$.
$\mathcal{M}, t, \bar{t} \models \neg A$ iff it is not the case that $\mathcal{M}, t, \bar{t} \models A$.
$\mathcal{M}, t, \bar{t} \models A \wedge B$ iff $\mathcal{M}, t, \bar{t} \models A$ and $\mathcal{M}, t, \bar{t} \models B$.
$\mathcal{M}, t, \bar{t} \models \mathcal{S}(A, B)$ iff there exists a $t' \in T$ with $t' < t$ and $\mathcal{M}, t', \bar{t} \models A$ and for every $t'' \in T$, whenever $t' < t'' < t$ then $\mathcal{M}, t'', \bar{t} \models B$.
$\mathcal{M}, t, \bar{t} \models \mathcal{U}(A, B)$ iff there exists an $t' \in T$ with $t < t'$ and $\mathcal{M}, t', \bar{t} \models A$ and for every $t'' \in T$, whenever $t < t'' < t'$ then $\mathcal{M}, t', \bar{t} \models B$.
$\mathcal{M}, t, \bar{t} \models \overline{\mathcal{S}}(A, B)$ iff there exists a $\bar{t}' \in T$ with $\bar{t}' < \bar{t}$ and $\mathcal{M}, t, \bar{t}' \models A$ and for every $\bar{t}'' \in T$, whenever $\bar{t}' < \bar{t}' < \bar{t}$ then $\mathcal{M}, t, \bar{t}'' \models B$.
$\mathcal{M}, t, \bar{t} \models \overline{\mathcal{U}}(A, B)$ iff there exists a $\bar{t}' \in T$ with $\bar{t} < \bar{t}'$ and $\mathcal{M}, t, \bar{t}' \models A$ and for every $\bar{t}' \in T$, whenever $\bar{t} < \bar{t}'' < \bar{t}'$ then $\mathcal{M}, t, \bar{t}'' \models B$.

Note that the semantics of horizontal and vertical operators are totally independent from each other, i.e. the horizontal operators have no effect on the vertical dimension and similarly for the vertical operators. If we consider the formula without the vertical operators, we have a one-dimensional horizontal $\mathcal{U}$ $\mathcal{S}$-temporal logic: similarly for the vertical temporal logic. Unary temporal predicates can be defined for both dimensions in the usual way, so we get $\Box$, $\blacksquare$, $\Diamond$, $\blacklozenge$, etc, for the horizontal dimension and $\overline{\Box}$, $\overline{\blacksquare}$, $\overline{\Diamond}$, $\overline{\blacklozenge}$, etc, for the vertical one.

3.2 Two-Dimensional Separation

The separation result does not hold for two-dimensional temporal logic above; i.e. , given a two-dimensional formula, it is not guaranteed that there exists an equivalent formula that is a conjunction of formulas of the form

$$\mathit{Past} \vee \mathit{Present} \vee \mathit{Future},$$

where *Present* contains no temporal operators, *Past* contains no future (i.e. $\mathcal{U}$ and $\overline{\mathcal{U}}$) operators and *Future* contains no past (i.e. $\mathcal{S}$ and $\overline{\mathcal{S}}$) operators. For example, the formula $\Diamond \overline{\blacklozenge}\, p$ cannot be separated.

However, for a very useful class of formulas called *temporalised formulas*, we obtain a restricted notion of separation, called *vertical separation*, which is strong enough for our purposes here.

A *temporalised formulas* is a two-dimensional formula in which no vertical temporal operator appears inside the scope of a horizontal temporal

operator. Eg. $\overline{\Diamond}\Diamond A$ is a temporalised formula, but $\Diamond\overline{\Diamond}A$ is not. Temporalised formulas can be seen as the result of having the vertical temporal dimension applied *externally* to a one-dimensional temporal logic, as discussed in [4].

A formula is *vertically separable* if it is equivalent to a formula that is a conjunction of formulas of the form

$$v\text{-}Past \vee v\text{-}Present \vee v\text{-}Future,$$

where *v-Past* contains no vertical future operators (i.e. $\overline{U}$ and its derived operators), *v-Future* contains no vertical past (i.e. $\overline{\mathcal{S}}$ and its derived operators) and no vertical operator occurs in *v-Present*. The following result was proved in [4].

THEOREM 3.1 *If A is a temporalised formula then A is vertically separable.*

Of course, a totally analogous horizontal separation can be obtained for formulas where the horizontal dimension is applied externally to the vertical one. In this work, vertical separation is emphasized because in our modelling of temporal database evolution (see Section 3.4), the horizontal dimension represents the state of the database, while the vertical dimension represents the evolution of such temporal database; hence it is the vertical dimension that is *external* to the database.

In this context, the horizontal dimension representing the temporal database state is called the *valid time* dimension. The vertical dimension representing the evolution of temporal database states is called the *transaction time* dimension.

3.3 The Two-Dimensional Diagonal

We now examine some properties of the diagonal in two-dimensional plane models. The diagonal is a privileged line in the two-dimensional model intended to represent the sequence of time points we call "now", i.e. the time points which an historical observer is expected to traverse. The observer is on the diagonal when he or she poses a query (i.e. evaluates the truth value of a formula) on a two-dimensional model.

So let δ be a special atom that denotes the points of the diagonal, which is characterised by the following property: for every $t \in T$ and every $\bar{t} \in \mathbb{Z}$:

$$\mathcal{M}, t, \bar{t} \models \delta \quad \text{iff} \quad t = \bar{t}.$$

The diagonal is illustrated in Figure 1.

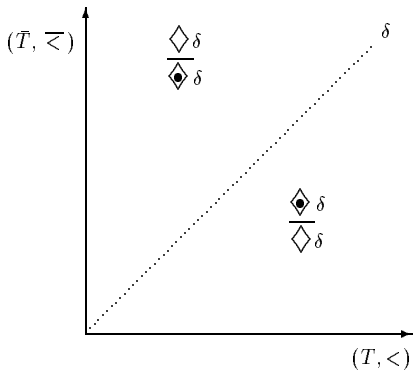

Figure 1 The two-dimensional diagonal

The following formulas are true at all points of the two-dimensional plane model:

$$\blacklozenge\,\delta \vee \delta \vee \Diamond\delta \qquad \blacklozenge\,\delta \leftrightarrow \overline{\Diamond}\delta$$
$$\delta \leftrightarrow (\,\Box\neg\delta \wedge \blacksquare\neg\delta \wedge \overline{\Box}\neg\delta \wedge \overline{\blacksquare}\neg\delta) \qquad \overline{\blacklozenge}\,\delta \leftrightarrow \Diamond\delta$$

The diagonal divides the two-dimensional plane in two semi-planes. The semi-plane that is to the (horizontal) left of the diagonal is "the past", and the formulas $\Diamond\delta$ and $\overline{\blacklozenge}\,\delta$ are true at all points of this semi-plane. Similarly, the semi-plane that is to the (horizontal) right of the diagonal is "the future", and the formulas $\blacklozenge\,\delta$ and $\overline{\Diamond}\delta$ are true at all points of this semi-plane. Figure 1 puts this fact in evidence.

The propositional approach of this section differs from the first order treatment of temporal features in the previous section. To reconcile these two different approaches a *propositional abstraction of database manipulations* can be developed. However, for space reasons, we omit such presentation, referring the reader to [7] for details.

3.4 Temporal Database Evolution

In describing the evolution of a temporal database, we have to distinguish the database evolution from the evolution of the world it describes. The "world", also called the *Universe of Discourse*, is understood to be any particular set of objects and relations between them in a certain environment that we may wish to describe. The database, in its turn, contains a description of the world. Conceptually, we have to bear in mind two distinct types of evolution, as introduced in [6]:

- The *evolution of the modelled world* is the result of changes in the world that occur independently of the database.
- A temporal database contains a description of the history of the modelled world that is also constantly changing due to database updates,

generating a sequence of database states. This *evolution of the temporal description* does not depend only on what is happening at the present; changes in the way the past is viewed also alter this historical description; moreover, changes in expectations about the future, if those expectations are recorded in the database, also generate an alteration of the historical description. This process is also called *historical revision*.

These two distinct concepts of evolution are reflected by a distinction between two kinds of flows of time, whether their time points refer to a moment in the history of the world, or whether they are associated with a moment in time at which a historical description is in the database.

Several different names are found in the literature for these two time concepts. The former is called *evaluation time* [13, 9], *historical time* [6], *valid time* [19] and *event time* [14]. The latter time concept is called *utterance time* [13], *reference time* [9], *transaction time* [6, 19] and *belief time* [20]. In this presentation we chose to follow a glossary of temporal database concepts proposed in [11], calling the former valid time, which is associated with the horizontal dimension in our two-dimensional model, and calling the latter transaction time, which is associated with the vertical dimension.

So we use the two-dimensional plane model to simultaneously cope with the two notions of time in the description of the evolution of a temporal database, as illustrated in Figure 2.

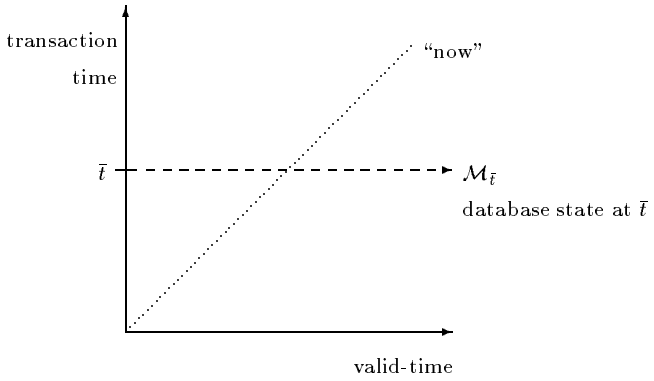

Figure 2 Two-dimensional database evolution

Let $\mathcal{M} = (T, <, \overline{T}, \overline{<}, \pi)$ be a two-dimensional plane model; its *horizontal projection* with respect to the vertical point $\bar{t} \in \overline{T}$ is the one-dimensional temporal model

$$\mathcal{M}_{\bar{t}} = (T, <, \pi_{\bar{t}}),$$

such that, for every propositional atom q, time points $t \in T$ and $\bar{t} \in \overline{T}$,

$$\pi_{\bar{t}}(t, q) = \top \quad \text{iff} \quad \pi(t, \bar{t}, q) = \top.$$

It follows that for every horizontal $\mathcal{U}$ $\mathcal{S}$-formula A and for every $t \in T$ and $\bar{t} \in \overline{T}$, $\mathcal{M}_{\bar{t}}, t \models A$ iff $\mathcal{M}, t, \bar{t} \models A$. The horizontal projection represents a state of a temporal database.

Updating temporal databases requires that, besides specifying the atom to be inserted or deleted, we specify its valid time. For that reason, it is convenient to use the notation of time-stamped atoms to represent the data being inserted and deleted. As a result, infinite updates are possible as long as such update is representable by a finite set of labelled formulas, where the label (time-stamp) is a temporal element. For example, $\Theta_- = \{[-\infty, +\infty] : p\}$ deletes the atom p for all times.

An *update pair* (θ_+, θ_-) consists of two finite disjoint sets of time-stamped atoms, where θ_+ is the *insertion set* and θ_- is the *deletion set*; by disjoint sets, in the context of data representation, it is meant that it is not the case that $\tau_+ : p \in \theta_+$ and $\tau_- : p \in \theta_-$ such that $\tau_+ \cap \tau_- \neq \varnothing$. We say that an update pair determines or characterises a *database update* $\Theta_{\bar{t}}$ occurring at transaction time $\bar{t} \in \mathbb{Z}$ if the application of the update function $\Theta_{\bar{t}}$ to the database state $\mathcal{M}_{\bar{t}} = (T, <, \pi_{\bar{t}})$ generates a database state $\Theta_{\bar{t}}(\mathcal{M}_{\bar{t}}) = (T, <, \Theta_{\bar{t}}(\pi_{\bar{t}}))$ satisfying, for every propositional atom q and every time point $t \in T$,

- if $\tau : q \in \theta_+$, then $\Theta_{\bar{t}}(\pi_{\bar{t}})(t, q) = \top$ for every $t \in \tau$;
- if $\tau : q \in \theta_-$, then $\Theta_{\bar{t}}(\pi_{\bar{t}})(t, q) = \bot$ for every $t \in \tau$;
- if neither $t : q \in \theta_+$ nor $t : q \in \theta_-$, then $\Theta_{\bar{t}}(\pi_{\bar{t}})(t, q) = \pi_{\bar{t}}(t, q)$.

The first item corresponds to the insertion of atomic information, the second one corresponds to the deletion of atomic information, and the third one corresponds to the persistency of the unaffected atoms in the database. Note that the disjoint sets θ_+ and θ_- are represented in the same way that the underlying database, so that we can represent a temporal database update schematically as:

$$\Theta_{\bar{t}}(\mathcal{M}_{\bar{t}}) = \mathcal{M}_{\bar{t}} \cup \theta_+ - \theta_-$$

When the sets θ_+ and θ_- are not disjoint, i.e. the update it is trying to insert and remove the same information, the situation is undetermined; typically this would mean that the transaction in which the update was generated should be rolled back. The update $\Theta_{\bar{t}}$ is a database state transformation function. An update may be empty ($\theta_+ = \theta_- = \varnothing$), in which case the transformation function is just the identity and the database state remains the same.

4 Imperative History

The two-dimensional temporal model can be applied in two distinct situations, namely:

- In the context of standard, one-dimensional temporal databases, the two-dimensional model is used to represent the evolution of the database. The current state of the database is constantly modified, i.e. history is constantly been rewritten, and past states of the database are not recorded.
- In the context of *bitemporal* databases, both dimensions are stored, so the two-dimensional model can be seen as modelling a state of the database.

In each of these two situations, the two-dimensional model allows us to lift the restrictions imposed by the imperative future to METATEM rules of the form

$$\text{past} \wedge \text{present} \rightarrow \text{future}$$

in a distinct way. In the context of bitemporal databases, this may lead to a two-dimensional imperative history over bitemporal databases, which is quite outside the scope of this presentation and is left as future work. So we concentrate one the first option, which is purely one dimensional.

4.1 Imperative History for a One-Dimensional Database

When only a single state representing the history of the world is stored in the database, an *imperative history* rule has the general format of

$$\text{history} \rightsquigarrow \text{history}$$

which, in formal terms is a formula of the form

$$\varphi \rightsquigarrow \psi$$

where both φ and ψ are any **FTL**-formulas that may refer to the (one-dimensional) present, past or future; φ and ψ are thus called *historical formulas*.

The meaning of an imperative history rule is, however, given in terms of the two-dimensional model. Therefore, the rule above is seen as representing the two-dimensional formula

$$\forall X \forall Y \, (\varphi \rightarrow \bar{\circ} \, \psi)$$

which clearly is a vertically separated formula. In this paper we will use such rules to specify that at every transaction time t, the corresponding formula holds at (t, t), i.e. on the diagonal.

This new view of **FTL**-formulas consists of the following reading of a temporal database evolution: "for all substitutions of the free variables that makes (the temporal query) φ true at the current time (i.e. on the diagonal point of the current state) force ψ to be true at this valid time in the next

state". The formula φ is called the *condition* or *query* part of the rule, and ψ is called the *action* part.

Several restrictions are imposed on the format of those rules. First, it is required that the set of free variables of ψ be free in φ so as to avoid undetermined actions. The complete rendering of an imperative history rule becomes:

$$\forall X \forall Y (\varphi(X,Y) \rightarrow \bar{\circ}\, \psi(X))$$

Second, it is required of the query part of the rule to be *range restricted*. This condition guarantees that queries have at most finitely many answers, and hence only finitely many actions to be executed. Range restrictedness demands that all the free variables occurring in the query formula should occur in a positive literal of the formula.

Finally, it is convenient to require that the action part of the rule be *deterministic* so that the rule management system will always know how to execute the rule. One way to avoid non-determinism is to constrain the format of the action formula to a conjunction of positive and negative literals, possibly preceded by a string of $\circ$'s or ◎'s. The following is a deterministic action formula:

◎`clear_top(X)` $\wedge$ $\neg$`occupied(X)` $\wedge$ $\circ\circ$`place(Obj, X)`

Note that in imperative history rules forcing ψ to be true may falsify φ. For instance, the simple rule $p \rightsquigarrow \neg p$ is legitimate, for it is understood as $p \rightarrow \bar{\circ}\, \neg p)$, i.e. whenever $t : p$ is true in the current state at transaction time t, $t : \neg p$ will hold at the next database state, not causing any inconsistency.

4.2 A System's Architecture to Support Imperative History

Conceptually, the support of imperative history rules is not much different from the support of standard rules found in non-temporal databases. As shown in figure 3, an architecture of such a system is centered on the temporal database manager (TDM), which performs all the tasks of a normal database manager, plus the manipulation of time.

Imperative history rules are stored in the rule manager (RM), and can be precompiled and optimized. The TDM activates the RM which, by selecting which rules to execute (see Section 6 on triggers), submit queries to the database via the TDM. The TDM then sends the answers back to the RM, and at the end of the rule execution cycle receives from it the actions that are to be performed. Some of these actions are temporal database updates which are sent to the database, others are messages sent to user or some kind of interaction with the temporal database's environment, such as an order to print a cheque or to signal an alarm, which are sent by the TDM via the environment interface module; others, still, may be triggering actions that will come back to the RM.

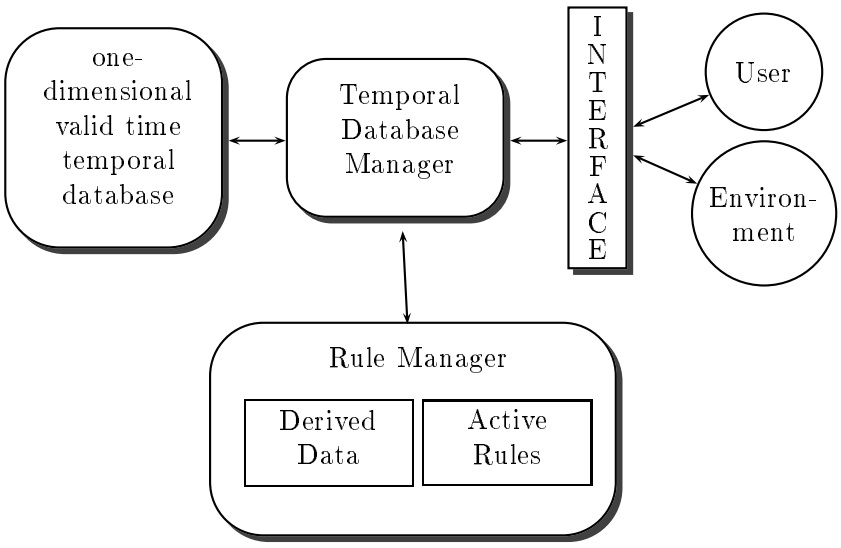

Figure 3. Possible architecture that supports IMPHIST rules

This apparently excessive traffic through the TDM in the processing of rules may be diminished if the RM is allowed to interact separately with the constituent parts of the TDM. In this way, the RM may interact directly with the database access module for querying and updating, receive triggers from and send triggers to the transaction selector, and send messages and requests to the environment directly to the interface module, avoiding the TDM in those cases. Of course, this is just a conceptual architecture, and each systems may adapt it differently to its own design philosophy.

5 Example

In this section we see the Imperative History ideas in action in a very simple example. The example concerns a patient monitoring system (PMS) for use in an intensive care ward of a hospital. It is based on the PMS system described in a software engineering context in [21] which has been implemented in a prototype version by an executable temporal logic language in [17]. The description and these implementations all concern a distributed system problem focussing on formal specifications of the communication between separate modules. Here we will introduce a historical dimension by requiring patient data to be recorded.

So suppose that we have a central nurse console (NC) interacting with several distributed patient monitors. The responsibilities of the NC will

include:

- recording information about a constant stream of patient data –here just heart rates– from the patient modules;
- recording data manually input by the nurse about changes in occupants of beds;
- notifying the nurse of alarming events (heart rate out of safety range etc);
- answering queries about beds and hearts in the present and the past;
- and correcting incorrectly recorded information.

A fully comprehensive database for this task would need to be two dimensional so that it could record information about the correction of past mistakes. Such information would be needed to explain actions which were taken on the grounds of information subsequently corrected. However, we will consider a simpler, one-dimensional temporal database adequate to hold a representation of the most up to date account of the history of the situation in the intensive care ward. Changes to data about the past will be allowed (as mistakes in recording do occur) but we will not necessarily be able to reconstruct the superseded model of the history of the ward.

The users will be primarily interested in the following two predicates:

`heart_rate(Patient,Rate)`	the heart rate of `Patient` is `Rate`
`occ(Bed,Patient)`	`Bed` is occupied by `Patient`

These are the most important predicates which vary in time in the world which the database will try to model. However it is easier to actually record some different predicates in the database. We introduce:

`hrm( Patient, Rate)`	the heart rate of `Patient` is measured as `Rate`
`chpat(Bed,Old,New)`	`Bed` is vacated by patient `Old` and occupied by patient `New`.

Empty beds can easily be handled within this formalism.

The simple (one-dimensional) temporal formula

$$\forall \texttt{p}.\ \forall \texttt{r}.\ \overline{\Box}\Box(\texttt{heart\_rate(p,r)} \Leftrightarrow \mathcal{S}(\texttt{hrm(p,r)}, \neg\exists \texttt{r2}.\ \texttt{hrm(p,r2)}))$$

specifies how to interpolate the most recent heart rate reading to any time. Below we will show how to render this in an imperative history rule.

Similarly, `chpat` and `occ` are related by

$$\forall \texttt{b}.\forall \texttt{p}.\overline{\Box}\Box(\texttt{occ(b,p)} \Leftrightarrow \mathcal{S}(\exists \texttt{q}.\texttt{chpat(b,q,p)}, \neg\exists \texttt{q'}.\ \texttt{chpat(b,p,q')})).$$

In general with deductive databases, whether temporal or otherwise, there is a distinction between basic and derived predicates. The database manager must be told which predicates are to have their history recorded. Other predicates play subsidiary roles: either being able to be derived from the recorded ones or appearing temporarily to produce, in combination with Imperative History rules, systematic changes in the database. In our example the database will only record the history of the predicates `hrm` and `chpat`.

Thus, the database manager will be responsible for

- translating automatically recorded heart rate measurements into database updates;
- doing the same for information entered by the nurse;
- answering queries which may be expressed in terms of derived predicates;
- producing the side-effect of an alarm which is described in terms of derived predicates;
- allowing error correction (including corrections of corrections e.t.c.) and disallowing nonsensical attempts at error correction.

Although our database is one dimensional, we use two dimensions of time to describe the way it changes over time. In this two dimensional approach, the predicates we have introduced take on a slightly expanded semantics: for example, `heart_rate(Patient,Rate)` holding at valid time t and transaction time $\overline{t}$ will mean that at transaction time $\overline{t}$, the database's model of the world contained the information that at valid time t, the heart rate of `Patient` is `Rate`.

Let us now examine some of the properties of this two-dimensional account and see which need explicit statement.

We suppose that automatic measurements arrive in the form of atoms t : `bhrm(b,r)` labelled with time instants. Such an atom indicates a heart rate measurement of `r` on the patient in bed `b` was made at the instant t. An atom like this may arrive at the manager at any time after t and so its effects will be recorded at some even later transaction time.

The effect of the arrival of such an atom is given by

$$\blacklozenge\,(\texttt{time(t)} \wedge \texttt{bhrm(b,r)} \wedge \texttt{occ(b,p)}) \rightsquigarrow \blacklozenge\,(\texttt{time(t)} \wedge \texttt{hrm(p,r)})$$

and the procedural effect of this rule is as follows. Suppose that at transaction time $\overline{t_1}$ the labelled atom t_2 : `bhrm(b,r)` is true. Then, the database manager will check, from the rule manager, whether t_2 : $\texttt{occ}(\texttt{b},\texttt{p})$ holds for any `p`. If so, then by transaction time $t_1 + 1$, the database will make $t_2 : \texttt{hrm}(\texttt{p},\texttt{r})$ hold.

Information entered by the nurse has a slightly more complicated effect. Suppose that the nurse enters the information that at time t, pa-

tient p vacates bed b and patient q replaces her or him. We use the atom nurse_requests_chpat_add(b,p,q,t) to indicate this. One of the effects of this information being entered is often the transaction

$$\blacklozenge\, \texttt{nurse\_requests\_chpat\_add(b,p,q,t)} \leadsto \blacklozenge\, (\texttt{time(t)} \wedge \texttt{chpat(b,p,q)}).$$

However we might want to disallow this direct update if the database currently shows that occ(b,p) is not true at that valid time. Thus we use

$$\texttt{nurse\_requests\_chpat\_add(b,p,q,t)} \wedge \texttt{occ(b,p)} \leadsto \blacklozenge\, (\texttt{time(t)} \wedge \texttt{chpat(b,p,q)}).$$

instead.

A Systematic Change to History

There is also another effect of the nurse entering a bed-change fact. Say that the nurse enters the information that at time t, the patient p is replaced by patient q in bed b. Any heart rate measurements already recorded from bed b but with later valid times than t are recorded as being about patient p but are now known to refer to patient q instead. Thus we have

$$\texttt{nurse\_requests\_chpat\_add(b,old,new,t)} \wedge \blacklozenge\, (\texttt{time(t)} \wedge \neg\texttt{chpat(b,old,new)}) \leadsto \blacklozenge\, (\texttt{time(t)} \wedge \texttt{chpat(b,old,new)})$$

This rule is a very good example of the power of two-dimensional temporal logic. It is a very clear case of a systematic change to history.

Queries and Side-effects

Queries are very straight forward to deal with. They are likely to be expressed in terms of occ and heart_rate and so can be answered using the derivation laws for these two predicates.

The syntax for the first of these rules is

$$\mathcal{S}(\texttt{hrm(p,r)}, \neg\exists \texttt{r2}.\, \texttt{hrm(p,r2)}) \leadsto \texttt{heart\_rate(p,r)}.$$

Note that there is no interesting two-dimensional character to this rule.

The side-effect of an alarm sounding for dangerous heart rate measurements can be produced by combining a derivation

$$\neg\, \mathcal{S}\, (\texttt{heart\_rate(p,r)} \wedge (\texttt{r} > \texttt{maxrate}),\ \texttt{alarm\_off(p)}) \leadsto\ \texttt{alarm!(p)}$$

with the hard-wiring of the current truth of predicate alarm! to the alarm bell. We associate an alarm sounding at time t with the predicate alarm! being true at time t in the model of the ward kept at transaction time t. The introduction of a nurse request to turn the alarm off and the corresponding predicate alarm_off is to prevent historical situations causing alarms.

Corrections

Corrections can not be handled by allowing the nurse to enter in directly the falsity of `chpat` for a particular historical time. This is because we must check for attempts at silly corrections. Thus we separate the fact of the nurse requesting a correction from the act of updating in accordance with that correction.

As well as corrections involving removal of recorded facts we also need to be able to add facts about the past. The nurse's requests are formalized in the predicate

$$\texttt{nurse\_requests\_hrm\_add(p,r,t)}$$

which means that the nurse requests that `hrm(p,r)` be added for valid time t and similar predicates `nurse_requests_hrm_remove(p,r,t)`, `nurse_requests_chpat_add(b,p,q,t)` and `nurse_requests_chpat_remove(b,p,q,t)`.

We introduce a predicate `error` whose truth at particular time on the diagonal has the side-effect of notifying the nurse that she or he has just requested a silly correction.

The rules which define the truth of `error` look like the following:

$$\texttt{nurse\_requests\_hrm\_add(p,r,t)} \wedge \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t)} \wedge \texttt{hrm(p,r)}) \rightsquigarrow \bigcirc(\texttt{error})$$

When `error` does not hold, then we use a set of Imperative History formulas to bring about the correction:

$$\begin{array}{l}\texttt{nurse\_requests\_chpat\_remove(b,old,new,t)}\wedge\\ \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t)} \wedge \texttt{chpat(b,old,new)}) \rightsquigarrow \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t)} \wedge \neg\texttt{chpat(b,old,new)})\\ \texttt{nurse\_requests\_chpat\_add(b,old,new,t)}\wedge\\ \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t)} \wedge \neg\texttt{chpat(b,old,new)}) \rightsquigarrow \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t)} \wedge \texttt{chpat(b,old,new)})\end{array}$$

and similarly for `hrm`.

Of course, just as with the nurse entering new information about bed changes, there is a consequence for heart rate measurements data from the correction of bed occupant data. We have

$$\begin{array}{l}\texttt{nurse\_requests\_chpat\_add(b,old,new,t1)}\\ \wedge\Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t2)} \wedge \texttt{hrm(old,r)} \wedge \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t1)} \wedge \neg\texttt{chpat(b,old,new)})\\ \rightsquigarrow \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t2)} \wedge \texttt{hrm(new,r)} \wedge \neg\texttt{hrm(old,r)})\\ \text{and}\\ \Diamond\!\!\!\!\blacklozenge\ (\texttt{nurse\_requests\_chpat\_remove(b,old,new,t1)}\\ \wedge\Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t2)} \wedge \texttt{hrm(new,r)} \wedge \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t1)} \wedge \texttt{chpat(b,old,new)}))\\ \rightsquigarrow \Diamond\!\!\!\!\blacklozenge\ (\texttt{time(t1)} \wedge \texttt{hrm(old,r)} \wedge \neg\texttt{hrm(new,r)})\end{array}$$

Correcting corrections turns out to be just the same as entering in new information.

Persistence

By requiring the database manager to follow the specification above— i.e. the explicit rules and the informal description of its operation— we actually end up abstractly building a two-dimensional temporal structure. A ground atomic formula is true in this structure at times $(t, \overline{t})$ if and only if the database claims at transaction time $\overline{t}$ that the formula is true at valid time t. It is clear that this structure is a model of the formulas explicitly mentioned above as rules.

However, the structure is also a model of many other formulas. For example, because of the operation of the database manager, only changing stored information in response to new data or entered requests, there are several *persistence* axioms. The one below indicates that heart rate measurements persist from an automatic measurement unless a request for change arrives:

$$\overline{\Box}\Box\forall \mathtt{p}.\ \forall \mathtt{b}.\ \forall \mathtt{r}.\ [\mathtt{hrm(p,r)} \Leftrightarrow \overline{\mathcal{S}}((\mathtt{bhrm(b,r)} \wedge \mathtt{occ(b,p)}) \vee \mathtt{nurse\_requests\_hrm\_add(p,r)}, \neg\mathtt{nurse\_requests\_hrm\_remove(p,r)})]$$

It is important to note that these two-dimensional rules describe the abstract model which is built by the database manager but they do not have to be explicitly programmed by the user.

Summary

In summary then, we can think of the running of the NC as the gradual construction of a two-dimensional temporal structure. Its language involves many predicates which have a variety of connections with the real world:

- `heart_rate` and `occ` affect the truth of queries;
- `hrm` and `chpat` represent the data stored;
- `alarm` and `error` produce side-effects;
- `nurse_requests_hrm_add` e.t.c. is true when the nurse makes a request to correct data via the NC;
- and `bhrm` is true when data arrives from a patient monitor.

There are also a variety of explicit or implicit properties exhibited by the structure:

- persistence arises from the inertia of the database;
- derivation rules define some predicates in terms of others;
- and Imperative History rules describe how various predicates require changes to the history recorded in the database.

6 Triggers

In this section we will enhance imperative history rules with *triggers*. Informally, a trigger is a mechanism that enables the processing of a rule. Only when a rule is enabled (i.e. when its trigger is fired) is its antecedent checked against the temporal database and the corresponding actions are executed; otherwise, when the trigger of a rule is not fired, the rule remains disabled and is ignored.

A trigger will be represented by a guard (i.e. a label) placed in front of the rule:

$$\mathit{trigger} : \mathsf{Condition} \rightsquigarrow \mathsf{Action}$$

and it is read as

when *trigger* is fired **if** Condition holds **then** execute Action.

A rule without a trigger can be thought of as being labelled by truth, $\top$, so it is always enabled.

One of the main reasons to include triggers in rules is to increase the system's efficiency. Without triggers, *all* rules have their antecedents checked against the database at every rule evaluation cycle — which, according to the two-dimensional model, means at every transaction time. When triggers are present, only the subset of enabled rules have their antecedent evaluated. We also say that the rule is *fired* when its trigger is.

Another good reason to include triggers is that they may be treated as channels through which the outside world communicates with the system, in the sense of [15]. For instance, in the rule sketched below:

$$\mathit{alarm\_ringing} : \mathsf{door\_open} \rightsquigarrow \mathsf{close\_door}.$$

the fact that the alarm is ringing is *external* to the system. This rule is activated only when it is communicated to the system that the alarm is ringing; therefore the trigger is called *external*. Only then it is checked whether the door is open.

Triggers may be also used in connection with an event *internal* to the database, such as the insertion $(+)$ or deletion $(-)$ of a fact. For example, when a door is recorded closed, we may wish to issue several warnings:

$$+\mathit{door\_closed} : \mathsf{emergency\_mode} \rightsquigarrow (\ \mathsf{lock\_door} \wedge \mathit{trigger\_lights}\ ).$$

As also seen in the example above, the Action-part of the rule can also fire an *internal* trigger (*trigger_lights*), which on its turn will activate another rule at the next transaction time. Therefore a *chain of control* of rule execution may be created through the rules.

Finally, there may be triggers related to events associated to a resource managed by the system, such as the system clock, file access, etc. For example:

$$\mathit{time\_is(8am)} : \mathsf{user\_logged\_in(User)} \rightsquigarrow \mathsf{good\_morning\_to(User)}$$

As we see, triggers can also carry parameters and have the same "appearance" as a normal database predicate.

There must be a mechanism for linking external and system triggers to their corresponding external and system events, but this will not be discussed here.

We will use the two-dimensional view of temporal database evolution to give a formal semantics to triggers. *Internal, external* and *system* triggers are seen as *non-persistent predicates*. We have to distinguish between trigger predicates and data predicates. When we defined the update semantics of data predicates in Section 3.4, whatever data was neither inserted nor deleted had its validity persisting into the following database state at the following transaction time.

However, trigger predicates have a "fixed duration" of a single transaction time unit. The insertion a trigger atomic predicate in the database corresponds to the *firing of a trigger* with the respective set of parameters. A trigger fired (i.e. inserted) at transaction time t will hold at the next transaction time, $t+1$. It will only hold at transaction time $t+2$ if has been re-fired (i.e. re-inserted) at $t+1$; otherwise it is removed from the database.

With this dynamic semantics for trigger predicates, we can construct *trigger expressions* by combining trigger predicates with boolean operators. A guarded rule of the form

$$\mathit{trigger\_expression} : \mathsf{Condition} \rightsquigarrow \mathsf{Action}$$

is then evaluated simply as

$$(\mathit{trigger\_expression} \wedge \mathsf{Condition}) \rightsquigarrow \mathsf{Action}.$$

The difference between *trigger_expression* and Condition remains solely in the dynamic behaviour of its components.

7 Conclusion

The main contribution of this paper is the suggestion of an appropriate executable temporal logic for declarative management of temporal databases. By using a two-dimensional temporal language and restricting our attention to a certain simple form of formula, we can express many useful patterns of updates and yet provide them with a direct procedural effect. We have demonstrated the usefulness of this language in a small example: in future work we hope to apply the language and develop the techniques in a much

larger example. It will be interesting to demonstrate the technique in action on a two-dimensional temporal database where the use of Imperative History rules in database management will be slightly different.

In other future work we hope to address questions of implementability of various restricted Imperative History languages.

References

1. James F. Allen. An interval based representation of temporal knowledge. In *Proc. 7th IJCAI*, 1981.
2. Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough, and Richard P. Owens. METATEM: A Framework for Programming in Temporal Logic. In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of *LNCS*, pages 94–129, Mook, Netherlands, 1989. Springer-Verlag.
3. Howard Barringer, Michael Fisher, Dov Gabbay, Richard Owens, and Michael Reynolds, editors. *The Imperative Future, vol. 1.* Research studies Press, 1996.
4. Marcelo Finger and Dov M. Gabbay. Adding a Temporal Dimension to a Logic System. *Journal of Logic Language and Information*, 1:203–233, 1992.
5. Marcelo Finger and Dov Gabbay. Combining Temporal Logic Systems. To appear in *Notre Dame Journal of Formal Logic*, 1996.
6. Marcelo Finger. Handling Database Updates in Two-dimensional Temporal Logic. *J. of Applied Non-Classical Logic*, 2(2):201–224, 1992.
7. Marcelo Finger. *Changing the Past: Database Applications of Two-dimensional Temporal Logics.* PhD thesis, Imperial College, Department of Computing, February 1994.
8. Michael Fisher and Richad P. Owens, editors. *Proceedings of IJCAI Workshop on Executable Modal and Temporal Logics, Chambery, France 1993*, volume 897 of *LNAI*. Springer-Verlag, 1995.
9. Dov M. Gabbay, Ian M. Hodkinson, and Mark A. Reynolds. *Temporal Logic — Mathematical Foundations and Computational Aspects, Vol. 1.* Oxford University Press, 1994.
10. Dov M. Gabbay. The Declarative Past and the Imperative Future. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Coloquium on Temporal Logic and Specifications — Lecture Notes in Computer Science 389*, Manchester, April 1987. Springer-Verlag.
11. Christopher S. Jensen, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass. A Glossary of Temporal Database Concepts. *SIGMOD RECORD*, 21(3):35–43, September 1992.
12. Hans Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, Michigan State University, 1968.
13. Hans Kamp. Formal Properties of Now. *Theoria*, 35:227–273, 1971.
14. L. Edwin McKenzie, Jr. and Richard T. Snodgrass. Evaluation of Relational Algebra Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4):501–544, December 1991.
15. Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
16. Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth Symposium of Principles of Programming Languages*, pages 179 – 190, 1989.
17. Mark Reynolds. Towards first-order concurrent metatem. In *[8]*. 1995.

18. Mark Reynolds. Axiomatising first-order temporal logic: Until and since over linear time. To appear in *Studia Logica*, 1996.
19. Richard T. Snodgrass and Ilso Ahn. A Taxonomy of Time in Databases. In *ACM SIGMOD International Conference on Management of Data*, pages 236–246, Austin, Texas, May 1985.
20. Suri Sripada. A Basis for Historical Deductive Databases. Internal report, Imperial College, Department of Computing, March 1990.
21. Wayne P. Stevens, Glenford Myers, and Larry L. Constantine. Structured design. *IBM. systems Journal*, 13(12):115–139, 1974.
22. Yde Venema. *Many-dimensional modal logic*. PhD thesis, University of Amsterdam, 1992.