PUC

Series:Monografias em Ciência da Computação

Nº 4/82

A TEMPORAL FRAMEWORK FOR DATABASE SPECIFICATIONS

José Mauro V. Castilho
Marco A. Casanova
Antônio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC / RJ - Departamento de Informática

Series: Monografias em Ciência da Computação, Nº 4/82

Editor : Marco A. Casanova Maio, 1982

A TEMPORAL FRAMEWORK FOR DATABASE SPECIFICATIONS*

José Mauro V. Castilho**
Marco A. Casanova
Antônio L. Furtado

* This work has been sponsored in part by FINEP and by CNPq grant 402090/80

** Universidade Federal do Rio Grande do Sul, Porto Alegre , RS - Brasil.

RESUMO:

Uma metodologia para descrição de bancos de dados é introduzida, incluindo o tratamento de restrições estáticas, isto é, restrições sobre que dados devem ser armazenados, bem como restrições de transição, isto é, restrições sobre como o banco de pode ser atualizado. Dois níveis de especificação são considerados. No primeiro nível, uma descrição D_1 de um banco de dados não indica como o banco será atualizado. Restrições de transição são então especificadas com o auxílio de uma variante conveniente de Lógica Temporal. No segundo nível de especificação, uma descrição D_2 de um banco de dados inclui um conjunto de operações pré-definidas que, por convenção, deverão ser usadas por qualquer transação que atualize o banco. Estas operações são descritas por suas propriedades e não por programas. Uma segunda variante de Lógica Temporal é usada para este propósito. As duas descrições, D_1 e D_2 são relacionadas por uma noção de refinamento, onde o conjunto de propriedades para as operações pré-definidas contido em D_2 deve garantir todas as restrições de D_1.

As vantagens obtidas através desta metodologia são de duas naturezas: descrições no primeiro nível contêm uma especificação estável e direta das restrições, enquanto que descrições no segundo nível sugerem uma estratégia efetiva para garantir as restrições.

PALAVRAS CHAVE:

Descrição de banco de dados, restrições de integridade, restrições de transições, correção de transações, Lógica Temporal, tipos de dados abstratos, encapsulamento.

Abstract:

A database description framework is introduced that accounts for static constraints, that is, constraints on what data can be stored, as well as transition constraints, that is, constraints on how data can be updated. Two levels of specification are considered. At the first level of specification, a database description D_1 does not indicate how the database will be updated. Transition constraints are then specified with the help of a convenient variant of Temporal Logic. By contrast, at the second level of specification, a database description D_2 includes a set of built in update operations which, by convention, must be used by any update transaction. These operations are described by their properties, rather than by actual code. A second variant of Temporal Logic is used for this purpose. The two descriptions, D_1 and D_2, are connected by a notion of refinement where the set of built-in update properties listed in D_2 must guarantee all constraints defined in D_1. The advantages accrued from this approach are twofold : first-level specifications give a direct, stable description of constraints, while second-level specifications suggest an effective strategy to enforce constraints.

Key Words:

Database description, integrity constraints, transition constraints, transaction correctness, Temporal Logic, abstract data types, encapsulation.

1. Introduction

We address in this paper the question of specifying databases that contain static constraints, that is, constraints on what data can be stored, and transition constraints, that is, constraints on how data can be updated. Moreover, any constraint may involve time. Examples of transition constraints are "salaries must never decrease" and " an employee that is currently assigned to a project cannot be fired" (i.e. he must first be disconnected from any project). Examples of constraints involving time are "an employee must receive a notice six weeks before being fired" and "no project can be inactive after January 1st, 1982".

We propose a multi-level database specification methodology, where levels differ essentially on how specific they are about database update operations. (Hence, our multi-level specification is orthogonal to the ANSI/SPARC three-level proposal [AN]). The first level of specification corresponds to the usual assumption that a database does not include any set of built-in update operations. Constraints on state transitions are described at this level with the help of a variant of Temporal Logic [RU]. For example, the constraint "salaries must never decrease" will be rephrased into a sentence whose intuitive meaning is "for any employee e, if e has now salary s, then in the future, if e is still an employee, he must have salary s', with $s' \geq s$". Notice that, in the last sentence,no verb such as "decrease", that suggests an operation, is used. Temporal circumstances are captured by timestamping the database or, putting it differently, by having an independent variable that stands for the "clock".

Temporal Logic has been successfully applied in a variety of problems, such as concurrent program verification [La,Pn,MP],network protocol specification [SM] , synthesis of communicating processes [MW], and information systems specification [Se] (the reference closest to our work).

At the second level of specification, a database description follows the idea of encapsulation [LZ]. That is, each database contains a predefined set of built-in update operations which, by convention,must be used by any update transaction. This strategy is advantageous because built-in updates can be designed so that no constraint is ever violated. Thus, users are relieved from worrying about consistency, because transactions will automatically preserve all constraints. This approach in no

way restricts queries, though.

We do not envision, at this second level of specification, built in updates described by actual programs. They are rather defined by their properties, which can take either the form of pre - and post- conditions [Ho,Br,Pa] or the form of equations which allow us to establish whether two sequences of operations will yield the same result [EKW,LM,VCF].

Although we will discuss only the two levels sketched above, we can easily imagine a third level where built-in updates are defined by actual programs.

Let D_1, D_2 and D_3 be the first, second and third level specifications of the same database. They establish a crescendo of abstraction in the sense that programs defined in D_3 are replaced by operation properties in D_2, which are in turn abstracted into operation-independent constraints in D_1.

This abstraction process must satisfy two properties, which embody a notion of refinement:

(a) programs defining built-in updates in D_3 must satisfy all properties listed in D_2;

(b) the set of built-in update properties listed in D_2 must guarantee all constraints defined in D_1 (assuming that state transitions can only be brought about by the built-in updates defined in D_2).

We close this discussion by briefly justifying our multi-level database specification methodology. The following points might be raised in favor of our approach:

(a) Since different repertoires of operations may span all (or part of the) valid states and valid transitions, first-level specifications tend to be more comprehensive and stable than second level specifications. We can often add or drop operations from a second level specification and yet stay within the bounds of the same first-level specification;

(b) Built-in operations are an effective way to enforce constraints, perhaps with the help of auxiliary structures. They can either be called by users' transactions or automatically, if considered as triggers [Es];

(c) both levels of specification are useful: first level specifications give a direct description of constraints, whereas second level specifications suggest implementation strategies, following the idea of encapsulation, that guarantee consistency preservation.

This last remark deserves additional comments. Let D_2 be a second level database description based on built-in operations. If we ask ourselves what set of database states D_2 specifies, the answer is: whatever the given set of operations happens to generate. Therefore, an independent definition of the set of acceptable states and of the allowed state transitions, as contained in a first level specification D_1, sounds superfluous.

Strictly speaking the above remark is indeed correct, but we believe that constraints cannot be ignored (see the remarks by Christian, Smith and Balzer in [BZ]). When programming methodologies are discussed, it is adequate to describe the behavior of data, which are accessory elements in a computation, by whatever the operations happen to do.However, data is fundamental for databases. Therefore, valid states and state transitions should be characterized independently of the set of built-in operations.

To further emphasize our point, consider the example, described in [VCF], of an employment agency database. In this example, a person can be hired through the agency only if he is a candidate to a job; after being hired, he ceases to be a candidate; a person becomes a candidate when he applies to the agency or when he is fired from a job he obtained through the agency. Note that these sentences actually describe properties of the built-in operations hire, apply and fire. While each of these properties is quite easy to understand, it may not be so obvious that, taken together, they are a way to enforce the simple constraint: "no person can simultaneously work on more than one company, if he obtains all his jobs through the agency".

A short description of each section now follows. Section 2 informally describes the scenario that underlies our formal treatment. Section 3 introduces the formalism we will use throughout the paper. Section 4 discusses first level database specifications, which include static and transition constraints. Section 5 addresses second-level database specifications (with built-in updates) and their relationship with first-

level specifications. Finally, Section 6 contains conclusions and directions for future research.

2. Informal Discussion of Basic Concepts

In this section we informally cover some basic concepts connected with database specifications. We begin by discussing the role of consistency criteria. We regard consistency criteria as describing policies of the enterprise and as disciplinating actions that produce or modify information about the enterprise. That is, an action is legal if and only if it preserves all consistency criteria.

The disciplining role of consistency criteria is greatly reinforced if we assume that actions take place only through the database. For example, we do not merely record that a person is hired by a company; a person is actually hired if and when the appropriate update is sucessfully executed in the database. Hence, no action that violates a consistency criterion can actually take place.

This assumption cannot always hold in practice, however. There are external actions performed, say, by government agencies that affect the enterprise. Thus, these actions will simply be recorded after notice is received that they took place. Examples are: tax cuts, price increases, etc...

A consequence of this assumption concerns the role time plays in consistency criteria. An unqualified reference to time would be ambiguous, since we would have real-world time and time as recorded by the internal clock of the system where the database is running. However, since we assumed that actions take place through the database, it is the clock time that counts. Or rather, the time of execution of an action is identified with the time when its results are recorded in the database. Hence, no "actuality lag" [Bu] exists.

We now examplify what we mean by static and transition constraints, which may or may not involve time. Suppose that we have a database with three tables or relation schemes [Da], WORKER[NAME], COMPANY [CNAME] and WORKSFOR[NAME, CNAME]. By convention, WORKER(w) indicates that person w is a certified worker; COMPANY(c) indicates that company c is active; and WORKSFOR(w, c) means that worker w works for company c.

In order to make our example quite explicit we assume that there is just one company C and one worker W. Then, the database has eight possible states as shown in Figure 3.1 (rows represent states and columns indicate the different values of each table).

	WORKER	COMPANY	WORKSFOR
(1)	∅	∅	∅
(2)	{W}	∅	∅
(3)	∅	{C}	∅
(4)	∅	∅	{(W,C)}
(5)	{W}	{C}	∅
(6)	{W}	∅	{(W,C)}
(7)	∅	{C}	{(W,C)}
(8)	{W}	{C}	{(W,C)}

Figure 3.1

It seems reasonable to impose the following static constraint:

s: "person p can work for company e only if p is a certified worker and e is an active company"

Any state is valid iff it does not violate the static constraints. Hence, states 1, 2, 3, 5 and 8 are valid (with respect to constraint s).

Whenever the database changes, we say that a state transition occurs. In our simple example, the set of all possible transitions can be represented by a digraph G = (N,E) whose nodes correspond to the five valid states and whose edges represent all possible state transitions(the reason for including loops (i,i) will be given later).

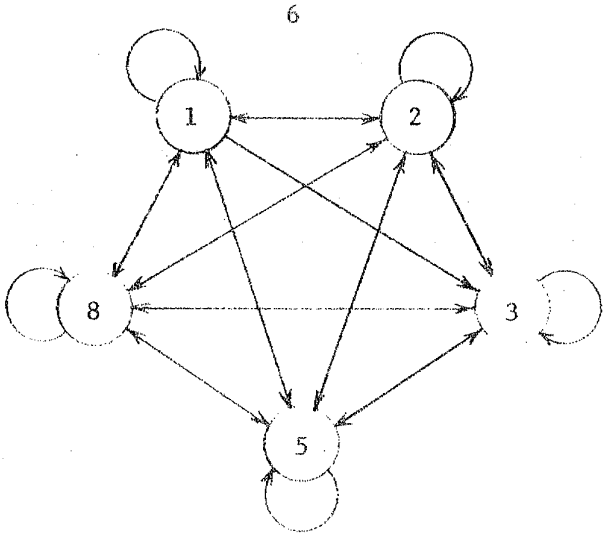


Figure 3.2

Consider now the following three transition constraints:

t1: "company e cannot become inactive if some person p works for e";

t2: "if person p stops working for company e, then p can never work for e again";

t3: "if worker p is unemployed for more than m units of time, then p ceases to be a certified worker (and, perhaps, becomes eligible to social security benefits)".

A set of state transitions is valid iff it satisfies all transitions constraints.

Thus, constraint t1 disallows transitions (8,1) and (8,2). However, from state 8, we can still reach a state where C becomes inactive, but only through a sequence of transitions, such as ((8,5),(5,2)). Therefore, constraint t1 imposes that the dismissal of W must be considered separately from, and performed prior to, the desactivation of C.

Constraint t2 disallows any sequence of transitions leading from state 8 to a distinct state and then back to state 8.

Constraint t3 says that transitions (2,1) or one of (5,1) and (5,3) become compulsory after the database has been evolving in or between states 2 and 5 for more than m units of time.

Thus, temporal circumstances , already implicit in constraints

t1 and t2 through the use of the adverbs after and again, are brought to the foreground in constraint t3. Note that we implicitly associated the flow of time with state transitions.This can be explicity done by assuming that:

(i) transitions are not instantaneous, that is, the value of the clock after a transition is strictly greater than the value of the clock before the transition;

(ii)conversely, the flow of time is always associated with some transition; if the database remains unchanged, then the trivial transition represented by a loop (i,i) is assumed (i.e. only the value of the clock changes).

Using the terminology of the Introduction, a first-level specification of our database then consists of the three relation schemas, WORKER[NAME], COMPANY[CNAME] and WORKSFOR[NAME,CNAME], the static constraint s and the transition constraints t1. t2 and t3.

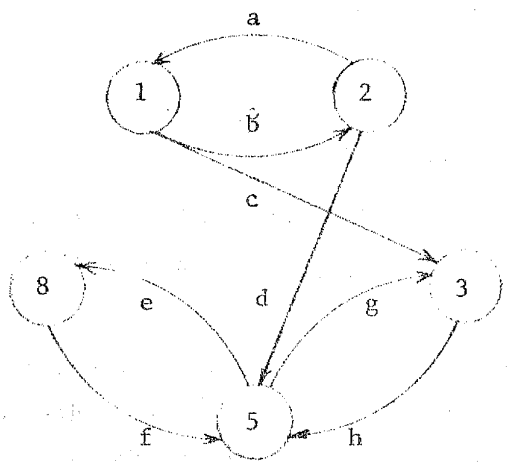
In order to pass to a second-level specification, we describe, via their properties, a set of built-in operations. The set we consider consists of the operations certify, register, hire, fire and cancel. The intended behavior of these operations is indicated in Figure 3.3, where they are shown as edge labels.

a) cancel(W)

b) certify(W)

c) register(C)

d) register(C)



e) hire(W,C)

f) fire(W)

g) cancel(W)

h) certify(W)

Figure 3.3

Note that the repertoire of operations chosen permits all valid states to be reached from the initial state 1, but it does not enable all valid transitions (e.g., there is no way to make C inactive). A set of operations is complete [SF] whenever any state is reachable from any other state through some sequence of updates. Hence, this property does not hold in our example and, in general, may or may not hold in a database application.

We briefly discuss only the hire and cancel operations. The intended effect of hire (W,C) is, of course, that WORKSFOR(W,C) becomes true. Thus, the second-level specification of our database would include the following property of hire:

h: "after hire (p,e) is successfully executed, WORKSFOR(p,e) becomes true".

However, the intended effect of hire must be disciplined so that no constraint is violated. Thus, to preserve the static constraints, hire(p,e) should fail on any state where either WORKER(p) or COMPANY(e) are false. This is captured by including the following property of hire in our second-level specification:

h': "if WORKER(p) or COMPANY(e) are false, then hire(p,e) must fail without modifying the database".

Consider now constraint t2: "if person p stops working for company e, then p can never work for e again". Any sequence of operations of the form

... hire(p,e);... fire(p);... hire(p,e);...

violates t2. Hence, an additional property of hire must be included in our second-level specification:

h'': "if WORKSFOR(p,e) was true in the past, then hire(p,e) must fail without modifying the database".

Since hire does not affect constraints t1 and t3 properties h, h' and h'' suffice to characterize hire and guarantee that no constraint is ever violated.

We observe at this point that sometimes it may be necessary to enchance the original database structures in order to define built-in operations that guarantee consistency preservation. For example, to guarantee property h'' above, it may be necessary to keep an extra table of former employees. At a more abstract level, we may assume that the data-

base is ever-growing [Su] in the sense that information about past states is always kept.

We now discuss the operation cancel. We begin by observing that the validity of constraints that involve time may not depend only on the way data is manipulated. Within this category, we find constraint t3: "if worker p is unemployed for more than m units of time, then p ceases to be a certified worker". In order to enforce such constraint, we introduce a special kind of built-in operation, cancel, called a trigger [Es]. The operation cancel will automatically delete any person p from WORKER as soon as p stops working for any company for more than m units of time. Thus, cancel will be activated independently of user's actions,unlike all other operations, which are called as part of the execution of a transaction. (Databases that can initiate action have been referred to as active databases in other contexts [MR]).

This concludes our remarks about constraints, built-in operations and multi-level specifications. The next sections formalize the concepts introduced here.

3. Basic Formalism

In this section we define a family of formal languages, called temporal languages, which are appropriate to describe both constraints on data and constraints on data transitions. We start by reviewing some basic concepts of first-order predicate calculus.

3.1 - First-Order Languages

We assume that the reader is familiar with the basic concepts of first-order predicate calculus [En,Sh]. So, we review very briefly only the concepts of many-sorted language, structure and first-order theory, mostly to set up some basic notation.

A many-sorted first-order language L [En,pp 277] is defined quite similarly to a first-order language, except that:

(i) L has a non-empty set S of sorts;

(ii) each variable belongs to a specific sort;

(iii) each n-ary predicate symbol p has an associated sort $(i_1,\ldots,i_n)$, which is a sequence in S;

(iv) each n-ary function symbol f also has an associated sort $(i_1,\ldots,i_n; i_{n+1})$;

(v) the formation rules of L respect sorts in the usual sense.

A structure I of L assigns to each sort i_j in S a domain D_{i_j}, to each n-ary predicate symbol r of sort $(i_1,\ldots,i_n)$ a relation $I(r)\subseteq D_{i_1}\times\ldots\times D_{i_n}$, and to each n-ary function symbol of sort $(i_1,\ldots,i_n; i_{n+1})$ a n-ary function $I(f)\colon D_{i_1}\times\ldots\times D_{i_n}\to D_{i_{n+1}}$. In particular, I assigns to each constant c of sort i_j an element $I(c)\in D_{i_j}$.

If a wff P of L is valid in a structure I of L, we write $\models_I P$. If P is valid in all structures of L, we write $\models P$.

A first-order theory is a pair $\sigma = (L,P)$ where L is a first-order language and P is a set of formulas of L, the non-logical axioms of σ. A model of σ is a structure of L where all formulas in P are valid.

3.2 Temporal Languages

We set up in this section a family of formal languages, called temporal languages, that permit expressing transition constraints such as "salaries never decrease" or "an employee cannot receive a raise during his first six months in the company". These transition constraints are interesting because they involve comparing data from different states (e.g., new salaries against old salaries). Examples of transition constraints formulated in temporal languages will be given in Sections 4.1 and 5.1.

Intuitively, a temporal language has objects of three types or sorts: (i) ind, that correspond to data elements; (ii) states that correspond to database states; and (iii) prog corresponding to programs.

We allow any function or predicate symbols over individuals. They will represent data structures or ordinary functions and predicates, such as '$\leq$'. However, function or predicate symbols of other sorts are restricted as follows. We will have one predicate symbol, after, of sort (prog, state, state). The intended interpretation of after(b,i,j) is that there is a computation of b that starts on state i and terminates on state j. We allow any n-ary function symbol f of sort (prog, ..., prog; prog) that creates a new program $f(b_1,\ldots,b_n)$ out of n programs $b_1,\ldots,b_n$. An example is the familiar program composition operation. We

also allow any n-ary function symbol g of sort (ind,...,ind; prog). The intended interpretation of after(g($\bar{x}$),i,j) is that there is a computation of procedure g that, when called with parameters $\bar{x}$, starts on state i and terminates on state j.

To relate objects of sort state to formulas, we add a new type of formula, $R_i(P)$, to the machinery of first-order logic. The intended interpretation of $R_i(P)$ is that the wff P holds on state i. We will also have a special symbol, cs, whose intended interpretation is that cs denotes the "current state". It will become clear that cs is neither a constant nor a variable. Hence, it constitutes a special characteristic of temporal languages. Both $R_i(P)$ and cs are taken, with minor modifications, from Temporal Logic [RU].

A few examples might help fix ideas at this point:

(1) $\forall i \forall j$ ($R_i(P) \wedge$ after (b,i,j) $\Rightarrow R_j(Q)$)
expresses that, if b starts on a state i satisfying P and terminates on a state j, then j satisfies Q;

(2) $\forall i \exists j$ ($R_i(P) \Rightarrow$ after (b,i,j))
says that, if the initial state satisfies P, then b always halts.

(3) $\exists i$ (after (b,cs,i) $\wedge R_i(P)$)
indicates that there is a computation of b that takes the current state into some state i where P is true.

More precisely, a temporal language *TL* is a typed language with three sorts: the individual sort (abbreviated ind), the state sort and the program sort (abbreviated prog). *TL* has the following symbols:

logical symbols:

connectives and parentheses: $\neg$, $\wedge$, (,)

variables: x,y,z,... of sort ind and
i,j,k,... of sort state

equality symbols: = of sort (ind,ind) and
$=_s$ of sort (state,state)

quantifiers : $\forall$ of sort ind and $\forall_s$ of sort state

special symbols: cs, the current state, and
R, the realization operator

non-logical symbols:

predicate symbols: for each $n>0$, there is a set (possibly empty) of n-place predicate symbols of sort (ind,...,ind)

special predicate symbol: a ternary predicate symbol, after, of sort (prog,state,state);

function symbols: for each $n\geq 0$, there is a set (possibly empty) of n-place function symbols of sort $(i_1,\ldots,i_n; i_{n+1})$, where $i_1,\ldots,i_{n+1}$ are either all of sort ind or all of sort prog, or $i_1,\ldots,i_n$ are all of sort ind and i_{n+1} is of sort prog.

note: for simplicity, we write $=$ and $\forall$ instead of $=_s$ and $\forall_s$.

We note at this point that we imposed restrictions on temporal languages that seem unnecessary. For example, there is no predicate symbol involving different sorts, except after, and no variables over programs (i.e., we cannot quantify over programs). These restrictions are justified by the scenario of this paper and could be relaxed in other situations.

The individual language induced by TL is the first-order language L whose symbols are those of TL, except the state variables, the symbols $=_s$, $\forall_s$, cs and R, the special predicate symbol after and all function symbols of sort $(i_1,\ldots,i_n; \text{prog})$, where i_j can be either ind or prog $(1\leq j\leq n)$. L is then the language used to talk about data.

The state language induced by TL is the first-order language T whose symbols are those of TL, except the special symbols cs and R and the symbols of L. T is then the language used to talk about states and programs. Indeed, TL can be viewed as combining T and L via the realization operator R and the current state cs.

The set of terms of TL is defined inductively as follows:

(1) any variable is a term;

(2) the special symbol cs is a term of sort state;

(3) if f is an n-ary function symbol of sort $(i_1,\ldots,i_n; i_{n+1})$ and $t_1,\ldots,t_n$ are terms of sort $i_1,\ldots,i_n$, respectively, then $f(t_1,\ldots,t_n)$ is a term of sort i_{n+1}.

note: the only terms of sort state are then the state variables themselves and the current state cs, since no function symbol involves the sort state.

The set of well-formed formulas (wffs) of *TL* is defined inductively as follows:

(1) if p is an n-ary predicate symbol of sort $(i_1,\ldots,i_n)$, with $i_j \in \{ind, state, prog\}$ $(1 \leq j \leq n)$, and $t_1,\ldots,t_n$ are terms of sort $i_1,\ldots,i_n$, respectively, then $p(t_1,\ldots,t_n)$ is a wff;

(2) if t_1, t_2 are terms of the individual sort (state sort), then $t_1 = t_2$ $(t_1 =_s t_2)$ is a wff;

(3) if P and Q are wffs, x is an individual variable and i is a state variable, then $\neg P$, $P \wedge Q$, $\forall x P$, $\forall_s i P$ are wffs;

(4) if P is a wff and u is a term of sort state, then $R_u(P)$ is a wff.

We also add to *TL*, by definition, the existencial quantifier $\exists$ and the boolean connectives $\vee$, $\Rightarrow$, $\equiv$, in the usual way.

Examples of wffs of *TL* are [RU]:

(1) $R_i(\neg P) \equiv \neg R_i(P)$

(2) $R_i(P \wedge Q) \equiv R_i(P) \wedge R_i(Q)$

(3) $R_i(\forall x P) \equiv \forall x\, R_i(P)$

(4) $R_i(\forall_s j P) \equiv \forall_s j (R_i(P))$, i and j distinct

(5) $R_{cs}(P) \equiv P$

(6) $R_i(R_j(P)) \equiv R_j(P)$

(7) $R_i(cs =_s j) \equiv i =_s j$

(8) $R_i(j =_s k) \equiv j =_s k$

(9) $\forall_s j(P) \Rightarrow P[cs/j]$, j does not occur within the scope of an R operator.

note: P[cs/j] indicates the result of replacing every free occurrence of j in P by cs.

The semantics of *TL* will be such that the wffs (1) through (9) are always true. Formulas (5), (6) and (9) deserve some comment. Formula

(5) captures the fact that cs is the current state. Formula (6) indicates that the result of $R_j(P)$ will not change when reevaluated in state i (formula (6) should be contrasted with $R_i(R_{cs}(P)) \equiv R_i(P)$, which is implied by (5)). Formula (9) indicates that, if P is true in every state j, then P is, in particular, true now (i.e. in the current state).

We now turn to the semantics of *TL*. A structure of *TL* is a quintuple $A = (I,S,P,A,B)$ where:

(a) I is a non-empty set of individuals, the individual domain of A;

(b) S is a non-empty set of states, the state domain of A;

(c) P is a non-empty set of programs, the program domain of A;

(d) A is a function assigning to each $a \in L$, a structure A_a of L, the individual language induced by *TL*, such that all structures have the same domain, which is I;

(e) B is a function assigning to after a relation $B(\text{after}) \subseteq P \times S \times S$, and to each n-ary function symbol f of sort (prog,...,prog; prog), $n \geq 0$, an n-ary function $B(f) : P^n \to P$, and to each m-ary function symbol g of sort (ind,...,ind; prog), $m \geq 0$, an m-ary function $B(g) : I^m \to P$.

Hence, associated with each state $a \in S$, there is a structure A_a giving meaning to the symbols of the individual language L induced by *TL*. The function B can also be viewed as a structure for the state language T induced by *TL*. Note then that the meaning of the symbols of T is fixed in A, whereas those of L have variable meaning.

We assign meaning to the wffs of *TL* as follows. Let v be a function that assigns to each individual variable x, an individual $v(x) \in I$ and to each state variable i, a state $v(i) \in S$. Given $a \in S$, we then define a function v_a, the extension of v in A, that assigns to each term t of sort s an element of the domain in A of sort s. The function v_a is defined inductively as follows:

(1) $v_a(u) = v(u)$, if u is a state or individual variable;

(2) $v_a(cs) = a$

(3) $v_a(f(t_1,\ldots,t_n)) = F(v_a(t_1),\ldots,v_a(t_n))$
where F is the function associated with f by A_a, or by B, if f is of sort $(i_1,\ldots,i_n;$ prog).

We now extend v_a to the wffs of TL as follows (the value of $v_a(P)$ will be true or false);

(1) $v_a(p(t_1,\ldots,t_n)) =$ true iff $(v_a(t_1),\ldots,v_a(t_n)) \in \mathcal{R}$

false otherwise

where $\mathcal{R}$ is the relation associated with p by A_a, or by B, if p is the special predicate symbol after;

(2) $v_a(t_1 = t_2) =$ true iff $v_a(t_1) = v_a(t_2)$

false otherwise

(3) $v_a(t_1 =_s t_2) =$ true iff $v_a(t_1) = v_a(t_2)$

false otherwise

(4) $v_a(\neg P) =$ true iff $v_a(P) =$ false

false otherwise

(5) $v_a(P \wedge Q) =$ true iff $v_a(P) = v_a(Q) =$ true

false otherwise

(6) $v_a(\forall x P) =$ true iff $v_b(P) =$ true, for every function v_b that agrees with v_a on all symbols, except on x

false otherwise

(7) $v_a(\forall_s i P) =$ true (same as in (6), except that i replaces x)

false otherwise

(8) $v_a(R_u(P)) =$ true iff $v_b(P) =$ true , where $b = v_a(u)$

false otherwise

Finally, we say that a wff P of TL is valid (written $\models P$) iff, for every structure $A = (I,S,P,A,B)$ of TL, for every assignment v of values (from I and S) to the variables of TL, and for every $a \in S$, $v_a(P) =$ true.

We conclude our list of basic definitions by saying that a structure $A = (I,S,P,A,B)$ of TL is standard iff P is a set of binary relations over S and, for any $\alpha \in P$ and any i, $j \in S$, $(\alpha,i,j) \in B$(after) iff $(i,j) \in \alpha$. We may consider, from now on, only standard structures of TL because the following lemma holds.

Lemma 3.1:

For any structure A = (I,S,P,A,B) with disjoint domains, there is a homomorphism h of A into a standard structure A' = (I,S,P',A,B') such that

(a) h is one-to-one on individual domain I and on the state domain S;

(b) for any assignment v of values to the variables of TL (from I and S), for every $a \in S$, and every wff P of TL,

$$v_a(P) = \underline{\text{true}} \quad \text{iff} \quad v'_a(P) = \underline{\text{true}}$$

where v_a, v'_a are the extensions of v in A and A' respectively. □

Lemma 3.1 tells us that, without loss of generality, we can always consider a program b as represented by a binary relation α such that $(i,j) \in \alpha$ iff there is a computation of b that starts on state i and terminates in state j.

This concludes the description of our basic formalism. The next sections discuss how it can be used to describe databases.

4. First Level Database Specifications

We discuss in this section how to describe databases at the first level of abstraction. That is, the descriptions will include constraints on state transitions, but they will not be based on any set of built-in update operations. Descriptions of this sort will use a class of temporal languages, called basic temporal languages.

4.1 - Basic Temporal Languages

A basic temporal language TL is a temporal language whose induced state language contains just two constants of sort prog, p and p*, and no other function symbol. The intended interpretation of after(p,i,j) is that j is obtained by applying some (unspecified) built-in operation to i. That is, j is an immediate successor of i, considering built-in operations as indivisible. The intended interpretation of after(p*,i,j) is that j is obtained by applying zero or more (unspecified) built-in operations to i. This is captured in part by defining a basic standard structure of TL as a standard structure A=(I,S,P,A,B) of TL such that $B(p^*)$ is the reflexive and transitive closure of $B(p)$ (recall that, since A is a standard structure, $B(p)$ and $B(p^*)$ are both binary relations).

Basic Temporal Languages give us a flexible and general mechanism to define tense operators or modalities, which increase the readability of formulas. Table 4.1 contains a list of classical modalities [RU], as well as two other modalities that we have found useful.

We close this section with a few examples that help assess the expressive power of basic temporal languages. Assume that *TL* has two binary predicate symbols, EMP and ASSIGN; the intended interpretation of EMP(n,s) is that employee n has salary s, and of ASSIGN(n,p) is that employee n is assigned to project p.

Using modalities, the sentence "salaries never decrease" could be formalised as:

(1) $\forall n\ \forall s\ \forall s'\ (EMP(n,s) \Rightarrow G^*(EMP(n,s') \Rightarrow s \leq s'))$

It is interesting to rephrase (1) in English again, since it says that "for any employee n, if n now has salary s, then henceforth if n has salary s', then $s \leq s'$. Thus, according to (1), if an employee is fired and hired again, his second salary must be greater than or equal to his first salary. That is, nowhere in (1) we expressed that the person must be continuously hired in order for the rule to apply.

If we understand "salaries never decrease" as saying that "during the lifetime of the same contract, the salary of an employee can not decrease", then we would have:

(2) $\forall n\ \forall s\ (EMP(n,s) \Rightarrow (\exists s'(EMP(n,s') \wedge s \leq s')$ while henceforth $\exists s''EMP(n,s'')))$

The need for two constants of sort prog, p and p*, with their intended interpretations, arose because we came across sentences that imposed restrictions on single state transitions, as opposed to restrictions on sequences of state transitions. For example, consider the sentence "employees that are assigned to a project cannot be fired". It could be formalized as:

(3) $\forall n\ (\exists p\ ASSIGN(n,p) \wedge \exists s\ EMP(n,s) \Rightarrow G^*(\exists s'EMP(n,s')))$

A closer inspection of (3) reveals that its meaning is "if employee n is now assigned to some project p, then henceforth n will always be an employee", which is too restrictive. A better formalization is:

(4) $\forall n\ (\exists s\ EMP(n,s) \wedge \exists p\ ASSIGN(n,p) \Rightarrow G(\exists s'\ EMP(n,s)))$

that restricts, under the intended interpretation, what built-in operations

Table 4.1

A List of Modalities

classical unary modalities for p

(1) $F(P) \equiv \exists i\,(\underline{after}(p,cs,i) \wedge R_i(P))$ - "eventually P after" (i.e., "eventually P holds after executing some built-in operation, starting on the current state)

(2) $P(P) \equiv \exists i(\underline{after}(p,i,cs) \wedge R_i(P))$ - "eventually P before"

(3) $G(P) \equiv \neg F(\neg P)$ - "always P after"

(4) $H(P) \equiv \neg P(\neg P)$ - "always P before"

classical binary modalities for p*

(5) $(P \ \underline{since}\ Q) \equiv \exists i(\underline{after}(p^*,i,cs) \wedge R_i(Q) \wedge \forall_j(\underline{after}(p^*,i,j) \wedge \underline{after}(p^*,j,cs) \Rightarrow R_j(P)))$

"P has been true up to now since Q was true"

(6) $(P \ \underline{until}\ Q) \equiv \exists i(\underline{after}(p^*,cs,i) \wedge R_i(Q) \wedge \forall_j(\underline{after}(p^*,cs,j) \wedge \underline{after}(p^*,j,i) \Rightarrow R_j(P)))$

"P will always be true from now on until Q is true"

classical unary modalities for p*

(7) $F^*(P) \equiv (\underline{true}\ \underline{until}\ P)$ "eventually P in the future"

(8) $P^*(P) \equiv (\underline{true}\ \underline{since}\ P)$ "eventually P in the past"

(9) $G^*(P) \equiv \neg F^*(\neg P)$ "henceforth always P"

(10) $H^*(P) \equiv \neg H^*(\neg P)$ "heretofore always P"

non-standard binary modalities for p*

(11) $(Q \ \underline{whileheretofore}\ P) \equiv \forall i(\underline{after}(p^*,i,cs) \wedge \forall j(\underline{after}(p^*,i,j) \wedge \underline{after}(p^*,j,cs) \Rightarrow R_j(P)) \Rightarrow R_i(Q))$

"necessarily Q will be true while heretofore P was always true"

(12) $(Q \ \underline{whilehenceforth}\ P) \equiv \forall i(\underline{after}(p^*,cs,i) \wedge \forall j(\underline{after}(p^*,cs,j) \wedge \underline{after}(p^*,j,i) \Rightarrow R_j(P)) \Rightarrow R_i(Q))$

"necessarily Q will be true while henceforth P will always be true"

can do (i.e., no single application of one built-in operation can at the same time disconnect an employee from all his tasks and fire him).

To talk about timestamps, such as the date an employee was hired, we assume that TL has an individual constant τ whose value in state i is intended to be the time state i was created. Then, the sentence "an employee cannot receive a raise until he has been working for the company for Δ units of time" is formalized as:

(5) $\forall n\, \forall s\, \forall t(EMP(n,s) \wedge \mathcal{H}\, (\neg \exists s'\, EMP(n,s')) \wedge \tau = t \Rightarrow$

$$((\tau \leq t + \Delta \Rightarrow EMP(n,s))\ \underline{\text{whilehenceforth}}\ \exists s'' EMP(n,s'')))$$

The antecedent of (5) expresses that employee n was hired at time t with salary s; the consequent of (5) says that the salary of n must be s at any time $\tau \leq t+\Delta$, if n has been continuously employed by the company.

4.2 First Level Relational Schemas

We now use the concepts developed in Section 4.1 to formalize the notion of first level relational database schema and related notions, such as consistent database state. Schemas at this stage will include constraints on state transitions, but they will not be based on any set of built-in update operations.

Before defining what we mean by a schema, it is worth noting that we classify all symbols of a schema into two sets. The first set contains all symbols, such as '=', whose intended interpretation is fixed. The second set includes all symbols whose meaning varies over time, which will be the relation names in the case of the relational model. Their meaning at a given point in time t comprises what is called the database state at t (however, for simplicity, our definition of database state also includes the meaning of all other symbols).

DEFINITION 4.1:

(a) A triple $\sigma = (TL, P, Q)$ is a first level relational schema iff

(i) TL is a basic temporal language with a distinguished set of predicate symbols $\rho = \{r_1, \ldots, r_n\}$, r_i of arity k_i $(1 \leq i \leq n)$, the relation names of σ ; (Let L be the individual language induced by TL in what follows).

(ii) P is a set of wffs of L, the static constraints (integrity constraints

or consistency criteria) of σ;

(iii) Q is a set of wffs of TL, the transition constraints of σ, such that each $Q \in Q$ is not first-order;

(b) A database state of σ is a structure of L;

(c) A consistent database state of σ is a database state I of σ such that each static constraint $P \in P$ is valid in I;

(d) A database universe U of σ is a set of database states such that: (i) all database states differ only on the values of the relation names; (ii) for each $I \in U$, for every relation name r and every relation $\mathcal{R}$ over the common domain such that r and $\mathcal{R}$ have the same arity, there is $J \in U$ such that I and J differ only on the value of r, which is $\mathcal{R}$ in J;

(e) A database history of σ is a structure $A=(I,S,P,A,B)$ of TL such that the range of A is a database universe of σ;

(f) A consistent database history of σ is a database history A of σ such that all transition constraints of σ are valid in A. □

note: if TL contains the special constant τ, then the definition of database universe must be modified so that states also differ on the value of τ and, for each $I \in U$, for each individual a, there is $J \in U$ such that I and J differ only on the value of τ which is a in J. □

Thus, the pair $\sigma' = (L,P)$ in Definition 4.1 is a first-order theory and a consistent database state of σ is just a model of σ'. The notion that the meaning of all symbols, except the relation names, is fixed is embodied in clause (i) of the definition of database universe (clause (ii) assures that, if the value of r is changed to $\mathcal{R}$, the new database state is still a member of the universe U). Finally, Q should be viewed as defining constraints on the allowed database state transitions.

5. Second Level Specifications

We now discuss how to define databases that include built-in operations. However, we assume that built-in operations are described by their properties, rather than by programs in a programming language. We first define, in Section 5.1, the class of temporal languages we will use and the concept of second level schemas. Then, in Section 5.2, we discuss the

relationship between first and second level database specifications. Finally, Section 5.3 indicates how to account for triggers.

5.1 - Temporal Languages with Procedures and Second Level Schemas

A temporal language with procedures is a temporal language TL with a set of function symbols of sort (ind,...,ind;prog), called procedures. If b is a procedure of TL, the term $b(\bar{x})$ is intended to denote a call to the procedure b with parameters $\bar{x}$.

A second level schema is a scheme $\sigma=(TL,P,Q)$ where TL is a temporal language with procedures. We also say that the procedures of TL are the built-in operations of σ.

We now briefly exhibit a second-level schema $\sigma = (TL,P,Q)$. We assume that TL has two binary predicate symbols, EMP and ASSIGN (see Section 4.1), a procedure raise with two parameters and two procedures with one parameter, fire and liberate. The static constraints will not be discussed, since they do not depend on new concepts. The transition constraints Q capture the intended interpretation of the procedures, which we now discuss.

The intended interpretation of raise(n,s) is that employee n receives a salary raise of s dollars, which is captured by adding the following wff to Q:

$$(1)\ \forall n\ \forall s\ \forall s'\ \forall i(EMP(n,s') \wedge s>0 \wedge \underline{after}(raise(n,s),cs,\ i) \Rightarrow R_i(EMP(n,s'+s)))$$

(Note the use of the current state symbol cs). The wff in (1) does not specify the behaviour of raise completely, though. Other wffs saying that raise(n,s) affects only the tuple of EMP corresponding to employee n, and no other tuple anywhere in the database, should be added to Q:

$$(2)\ \forall m\ \forall r\ \forall n\ \forall s\ \forall i(EMP(m,r) \wedge m \neq n \wedge \underline{after}(raise(n,s),cs,i) \Rightarrow R_i(EMP(m,r)))$$

$$(3)\ \forall m\ \forall p\ \forall n\ \forall s\ \forall i(ASSIGN(m,p) \wedge \underline{after}(raise(n,s),cs,i) \Rightarrow R_i(ASSIGN(m,p)))$$

The intended interpretation of fire(n) is that employee n is dismissed. However, we wish to guarantee the constraint (see Section 4.1): "if an employee is assigned to some project, then he cannot be fired". Thus, Q must include the following wffs:

(4) $\forall n\ \forall s\ \forall i(EMP(n,s) \wedge \neg \exists p\ ASSIGN(n,p) \wedge \underline{after}(\underline{fire}(n),cs,i)$
$\Rightarrow R_i(\neg \exists s'\ EMP(n,s')))$

(5) $\forall n\ \forall s\ \forall i(EMP(n,s) \wedge \exists p\ ASSIGN(n,p) \wedge \underline{after}(\underline{fire}(n),cs,i)$
$\Rightarrow R_i(EMP(n,s)))$

As for raise, we must complete the specification of fire by adding to Q constraints saying that fire(n) only deletes from EMP the tuple corresponding to employee n (which we omit for brevity).

The intended interpretation of liberate(n) is that employee n is liberated from all tasks he has on all projects, which is captured by adding to Q the following wff:

(6) $\forall n\ \forall i\ (\underline{after}(\underline{liberate}(n),cs,i) \Rightarrow R_i(\neg \exists p\ ASSIGN(n,p)))$

Again, the specification of liberate must be completed by adding other wffs to Q as for raise.

This concludes our discussion about second-level schemas.

5.2 - The Relationship between First and Second Level Schemas

The first and second level specifications of the same database are not at all unrelated. In fact, we imagine that the second level specification is obtained from the first level description by a refinement process where: (i) a set of built-in operations is selected; (ii) built-in operation properties are introduced so as to guarantee all static and transition constraints defined in the first-level specification.

This section then defines precisely what we mean by refinement.

Let $\sigma'=(TL',R',Q')$ be a first level schema and $\sigma=(TL,R,Q)$ be a second level schema. We say that TL is a refinement of TL' iff

(i) TL' and TL differ only on the relation names of σ' and σ and on the program symbols of sort $(i_1,\ldots,i_2;\ \underline{prog})$ (i.e., TL' has two constants of sort prog, by definition, while TL has a set of function symbols of sort $(\underline{ind},\ldots,\underline{ind};\ \underline{prog})$);

(ii) there is a refining function γ assigning to each n-ary relation name r of TL' a wff r^γ of TL with n free individual variables ordered $x_1,\ldots,x_n$. Intuitively, γ defines each relation name r of TL' in terms of those of TL.

Suppose that TL is a refinement of TL' with refining function γ.

Given a standard structure $A=(I,S,P,A,B)$ of TL, we construct a standard structure $A^\gamma=(I,S,P',A',B')$ of TL' as follows. P' contains just two elements, α and α^*, where $\alpha = \bigcup_{\beta \in P} \beta$. B' is such that $B'(p) = \alpha$ and $B'(p^*) = \alpha^*$ (B'(after) is then fixed, by definition of standard structure). For each $a \in S$, A'_a is equal to A_a, except on the value of each relation name r of σ', where $A'_a(r)$ is the n-ary relation defined by r^γ in A_a. A^γ is called the structure induced by A; likewise, A'_a is the database state of σ' induced by A_a.

Intuitively, the value of p in A^γ, $B'(p) = \alpha$, contains all state transitions that can be brought about by a call to some built-in operation of σ; likewise, the value of p^* in A^γ contains all state transitions that can be brought about by zero or more calls to built-in operations of σ. Hence A^γ assigns the intended interpretation (in terms of the built-in operations of σ) to p and p^*. The value of each relation name r of σ' in each A'_a is obtained from A_a via r^γ, the wff defining r in TL.

Finally, given a first-level schema $\sigma'=(TL',R',Q')$ and a second level schema $\sigma=(TL,R,Q)$, we then say that σ is a refinement of σ' iff: (i) TL is a refinement of TL' with refining function γ; (ii) for any standard structure A of σ that satisfies Q, A^γ also satisfies Q'; (iii) for any consistent database state I of σ, the induced state I^γ of σ' is also consistent. Thus, condition (ii) captures the idea that the constraints on the behaviour of built-in operations defined in σ suffice to guarantee the transition constraints of σ'. Similarly, condition (iii) implies that the static constraints of σ are enough to guarantee those of σ'.

We now indicate very briefly how we could prove that a second-level schema $\sigma=(TL,R,Q)$ is a refinement of a first-level schema $\sigma'=(TL',R',Q')$. Let γ be the function refining TL' into TL. For each wff $P \in R'$ we can define a wff P^γ of TL such that P is valid in I^γ iff P^γ is valid in I, for any database state I of σ (see, for example, [CCF]). Hence, testing condition (iii) reduces to proving that, for each $P \in R'$, P^γ is a logical consequence of R. Since P^γ is first-order, this presents no novelties.

Condition (ii) can be checked in much the same way as condition (iii). To illustrate this remark, suppose that σ is the schema described in Section 5.1, which has built-in operations raise, fire and liberate. We sketch how we could prove that the properties of these built-in operations listed in Q guarantee the constraint "if an employee is assigned to some

project, then he cannot be fired" (which might be considered a transition constraint of Q' for the purposes of this illustration).

This constraint was defined in Section 4.1 (in the basic temporal language TL') as follows:

(1) $\forall n\ \forall s(EMP(n,s) \wedge \exists p\ ASSIGN(n,p) \Rightarrow G(EMP(n,s)))$

Expanding the definition of G, we would have:

(2) $\forall n\ \forall s(EMP(n,s) \wedge \exists p\ ASSIGN(n,p) \Rightarrow \forall i(\underline{after}(p,cs,i) \Rightarrow R_i(EMP(n,s)))$

Let A be a structure of TL and A^γ be the structure of TL' induced by A. Recall that the interpretation of $\underline{after}(p,cs,i)$ in A^γ was that state i can be reached from the current state cs by executing some built-in operation. That is, $\underline{after}(p,cs,i)$ holds in A^γ iff the wff P holds in A, where P is:

(3) $\exists n\ \exists s\ (\underline{after}(\underline{raise}(n,s),cs,i)) \vee$
$\exists n\ (\underline{after}(\underline{fire}(n),cs,i)) \vee$
$\exists n\ (\underline{after}(\underline{liberate}(n),cs,i))$

Then , the wff in (2) holds in A^γ iff the wff Q holds in A, where Q is:

(4) $\forall n\ \forall s(EMP(n,s) \wedge \exists p\ ASSIGN(n,p) \Rightarrow \forall i(P \Rightarrow R_i(EMP(n,s)))$

Therefore, given any structure A of TL that satisfies Q, the wff in (2) is valid in A^γ iff the wff in (4) is valid in A. But this is equivalent to proving that (4) is a logical consequence of Q, which can be established by an adaptation of the standard axiom system for temporal logic [RU].

5.3 - Triggers

Recall from Section 2 that a trigger p_i was an operation executed automatically whenever a certain condition B_i was true. We show in this section that we can view triggers as a method of defining built-in operations and that we need not leave the framework developed in Sections 5.1 and 5.2.

Let σ be a second level schema and suppose that operations $p_1,\ldots,p_m$ of σ are considered as triggers and operations $q_1,\ldots,q_n$ are treated as normal built-in operations. Let B_i be the condition for executing p_i, $1 \le i \le m$.

Our approach is indeed quite simple. We assume that the system will implement triggers by checking whether any trigger can be executed only before or after the execution of a normal built-in operation. We also assume that, if more than one trigger can be executed, the order of execution is

selected nondeterministically. These assumptions are equivalent to transforming each built-in operation q_i into the program $\bar{p}$; q_i; $\bar{p}$, where $\bar{p}$ = do $B_1 \rightarrow p_1 \,\square \ldots \square\, B_m \rightarrow p_m$ od (see [Di] for the guarded do-loop).

Thus, triggers can be viewed as a way of implementing complex built-in operations. We include $\bar{p}$ before and after q_i because a guard B_k may be affected by the results of q_i and, hence, must be tested after q_i executes. Moreover, after executing a built-in operation q_j and before executing the next one, q_i, the clock may advance thus rendering some guard B_k true, which was false right after the execution of q_j. Hence, each guard must be tested before q_i executes. (Note that, in our formalization, triggers are called only when a built-in operation is invoked).

Hence, to account for triggers, it remains to phrase $\bar{p}$ as a program of temporal languages with procedures. In order to do so, we first write $\bar{p}$ as a regular program [Ha]:

(1) $\bar{p} = (B_1?;\ p_1 \cup \ldots \cup B_m?;\ p_m)^*;\ (\bigwedge_{i=1}^{m} \neg B_i)?$

The union operation can be introduced in a temporal language as a binary function symbol, $\cup$, of sort (prog,prog; prog); a test B? can be introduced as constant of sort prog, for each wff B of TL; finally the reflexive and transitive closure operation can be introduced as an unary function symbol, *, of sort (prog; prog). The intended meaning of these symbols is captured by forcing, in each standard structure $A = (I,S,P,A,B)$, $B(\cup)$ to be the set theoretic union operation, $B(*)$ to be the set theoretic reflexive and transitive closure, and $B(B?) = \{(a,a) \in S^2 / \models_{A,a} B\}$. With these provisos, $\bar{p}$ becomes a term of TL of sort prog.

Then, we can reduce our second level schema σ with triggers to a similar one, σ', without triggers, simply by replacing each normal built-in operation q_i of σ by $\bar{p}$; q_i; $\bar{p}$. This reduction justifies our initial remark that we view triggers as a method of defining built-in operations.

6. Conclusions

We introduced a family of high level database specification languages, called temporal languages, that permit expressing transition constraints as well as the more traditional static constraints. These languages are quite flexible and permit defining a meaningful dictionary of modalities that greatly enhance the readability of transition constraints, without ad-

ditional conceptual burden.

We introduced the idea of multi-level database specifications of increasing degree of abstraction. At the highest level of abstraction, static and transition constraints are specified without mentioning any set of built-in operations. This description serves mostly to document the intended behavior of the database. That is, it describes both the nature of the data kept in the database and the rules governing how to create and modify such data.

Walking towards implementing the database, a set of built-in operations, that are able to create and modify the data kept, is identified. The database schema at this second level of description will include, besides the names of the operations, their properties. But no actual code for the operations is provided. The properties must be carefully defined so as to guarantee that no constraint listed in the first-level description is violated.

These were the two levels discussed in this paper. The trend would continue by choosing an implementation for the operations, which would require defining an appropriate class of temporal languages that include a suitable programming language (one such class was sketched at the end of Section 5.3; see also [CB]). More generally, going towards an implementation involves selecting a data structure oriented data model that contains, not only a programming language, but also suitable features to represent data.

Further theoretical work would involve investigating the decision problem for each class of Temporal Languages, starting with the definition of consistent and complete axiom systems for these languages. It is also worth investigating the expressive power of the modalities introduced in Section 4.1, vis-a-vis the necessities of high-level database specifications.

References

[AN] "Study Group on Data Base Management Systems: Interim Report" FDT 7.2, ACM(1975)

[Br] M.L.Brodie. "On Modelling Behavioural Semantics of Databases". Proc. 7th Int. Conf. on Very Large Data Bases (1981), 32-42

[Bu] J. Bubenko. "The Temporal Dimension in Information Modelling". In "Architecture and Models in Data Base Management Systems (ed. by G. Nijssen). North-Holland (1977)

[BZ] M.L. Brodie, S.N. Zilles. (Discussion about Consistency of Models, page 72, bottom of second column). Proc. Workshop on Data Abstraction, Databases and Conceptual Modelling(1981).

[CB] M.A. Casanova, P.A. Bernstein. "A Formal System for Reasoning about Programs accessing a Relational Database" ACM TOPLAS 2,3 (July 1980), 386-414

[CCF] M.A. Casanova, J.M.V. de Castilho, A.L. Furtado. "Properties of the Conceptual and External Schemas." Proc. Formalization of Programming Concepts (to appear)

[Da] D.J. Date. "An Introduction to Database Systems." (3rd Ed.) Addison-Wesley (1981)

[Di] E.W. Dijkstra. "A Discipline of Programming", Prentice-Hall (1976)

[En] H.B. Enderton. "A Mathematical Introduction to Logic." Academic Press (1972)

[Es] K.P. Eswaran. "Specification, Implementation and Interaction of a Trigger Subsystem in an Integrated Data Base System". IBM Research Report RJ1820 (Aug. 1976)

[EKW] H. Ehrig, H.J. Kreowski, H. Weber "Algebraic Specification Schemes for Database Systems." Proc. 4th VLDB Conf., Berlin (1978)

[Ha] D. Harel. "First-Order Dynamic Logic." Lecture Notes in Computer Science, vol. 68 (1979)

[Ho] C.A.R. Hoare. "An Axiomatic Basis for Computer Programming." CACM 12,10 (Oct. 1969)

[La] L. Lamport. "On the Temporal Logic of Programs" Technical Report CSL-86, SRI International (1979)

[LM] P.C. Lockemann, H.C. Mayr, W.H. Weil, W.H. Wohlleber. "Data Abstractions for Data Base Systems." ACM/TODS 4 (1979)

[LZ] B. Liskov, S. Zilles "Specification Techniques for Data Abstractions." IEEE Trans. on Soft. Eng. SE-1 (1975)

[Ma] Z. Manna. "Verification of Sequential Programs: Temporal Axiomatization." Report n. STAN-CS-81-877, Department of Computer Science, Stanford University (Sept. 1981)

[MR] C.A. Montgomery, E.H. Ruspini. "The Active Information System: a Data-Driven System for the Analysis of Imprecise Data." Proc. 7th Int. Conf. on Very Large Data Bases (1981), 376-384

[MP] Z. Manna, A. Pnueli. "Verification of Concurrent Programs, Part I: The Temporal Framework." Report n. STAN-CS-81-836, Department of Computer Science, Stanford University (June 1981)

[MW] Z. Manna, P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications." Report n. STAN-CS-81-872, Department of Computer Science, Stanford University (Sept. 1981)

[Pa] P. Paolini. "Verification of Views and Application Programs." Proc. Workshop on Formal Bases for Databases, Toulouse (1979)

[Pn] A. Pnueli . "The Temporal Logic of Programs." Proc. 18th Foundations of Computer Science Conference (Nov. 1979), 46-57

[RU] N. Rescher, A. Urquhart. "Temporal Logic." Springer-Verlag (1971)

[SF] K.C. Sevcik, A.L. Furtado. "Complete and Compatible Sets of Update Operations." Proc. ACM ICMOD (1978), 247-260

[Se] A.Sernadas . "Temporal Aspects of Logical Procedure Definition." Info. Systems 5 (1980), 167-187

[Sh] J.R. Shoenfield. "Mathematical Logic." Addison-Wesley (1967)

[Su] B. Sundgren. "An Infological Approach to Data Bases". Urval 7. SCB - Statistika Centralbyran. Stockholm (1973)