A General Treatment of Dynamic Constraints

E.O. de Brock

November 1997

SOM theme A: Intra-firm coordination and change

Abstract

This paper introduces a general, formal treatment of dynamic constraints, i.e., constraints on the state *changes* that are allowed in a given state space. Such dynamic constraints can be seen as representations of "real world" constraints in a managerial context. The notions of transition, reversible and irreversible transition, and transition relation will be introduced. The link with Kripke models (for modal logics) is also made explicit. Several (subtle) examples of dynamic constraints will be given. Some important classes of dynamic constraints in a database context will be identified, e.g., various forms of cumulativity, non-decreasing values, constraints on initial and final values, life cycles, *changing* life cycles, and transition and constant dependencies. Several properties of these dependencies will be treated. For instance, it turns out that functional dependencies can be considered as "degenerated" transition dependencies. Also, the distinction between primary keys and alternate keys is reexamined, from a dynamic point of view.

Keywords: Dynamic constraints, transition (relation), (ir)reversibility, transition dependencies, constant dependencies, cumulativity, (changing) life cycles, Kripke models

E.O. de Brock
Faculty of Management and Organization
State University of Groningen
P.O. Box 800, 9700 AV Groningen
The Netherlands
Tel. +31.50.3637315
Fax +31.50.3632275
Email: e.o.de.brock@bdk.rug.nl

Introduction

In data modelling, constraints play an important role. We can distinguish between static and dynamic constraints. The requirements on the *states* are called *static constraints*, and the requirements on the *state transitions* are called *dynamic constraints*. Traditionally, most emphasis in the constraints literature is on *static* constraints. This paper is dedicated to a general, formal treatment of *dynamic* constraints.

Although there exist many publications on temporal databases in the literature (see for example the bibliography in [Kl 93]), the related topic of dynamic constraints received much less attention. The papers on dynamic constraints usually concentrate on certain standardized forms of dynamic constraints (see [Vi 87], [Na 89], [Wij 95], and [Je 94], for instance). In our paper, we address (and solve) the problem of the development of a general theory of dynamic constraints in which all kinds of dynamic constraints can be neatly treated. We will also illustrate the richness and relevance of our theory with various (classes of) practical examples.

From a management point of view, it is sometimes inconvenient to treat dynamic constraints as "ironclad rules". After all, the situation might arise that in the end a certain transaction appeared to be erroneous or unintentional, while it cannot be undone because of the dynamic constraints. We say in this case that the transition is irreversible; if it can be undone, the transition is called reversible. In many cases of non-reversible transition relations, a dynamic constraint is not used as a "law" but rather as a warning ("Do you *really* want this?") or as a condition pertaining to special authorization required to carry out the modification.

All in all, this paper contributes to both the theory and application of dynamic constraints, among others by establishing a general theoretical foundation for solutions to practical problems, even in subtle and complex situations.

The paper is organized as follows. The notions of transition, reversible and irreversible transition, and transition relation will be introduced in Section 1. The link with Kripke models (for modal logics) is also made explicit here. Several examples of dynamic constraints will be given in Section 2. Here, the distinction between primary keys and alternate keys is also reexamined, from a dynamic point of view. The special classes of transition and constant dependencies will be identified in Section 3. Several properties of these dependencies will be treated here. In the Appendix our basic notions and notations are established.

1. Transition relations and (ir)reversibility

By defining a database universe U (or a state space in general), we establish which *states* are formally allowed in an organization. In general, however, not all state *transitions* between those (in itself) allowed states are admissible in that organization. We can establish the set of admissible transitions by means of a subset R of $U \times U$, having the following intuitive meaning:

$(v;v') \in R \Leftrightarrow$ the direct transition from state v to state v′ is allowed.

If $(v;v') \notin R$ then it might still be possible that v′ is *indirectly* reachable from v, namely if v′ is reachable from v via a number of allowed intermediate steps (or formally, if $(v;v') \in Tcl(R)$, where **Tcl(R)** denotes the transitive closure of R).

An element of $U \times U$ is called a *transition within* U. A set of admissible transitions for the state space U is called a *transition relation on* U. Formally:

Definition 1:
If U is a set, then:
(a) p is a **transition within** U $\Leftrightarrow$ $p \in U \times U$;
(b) R is a **transition relation on** U $\Leftrightarrow$ $R \subseteq U \times U$.

In summary, the requirements on the *states* in an organization are called *static constraints* and determine a state space U, while the requirements on the *transitions* are called *dynamic constraints* and determine a transition relation R on U.

The readers who are familiar with Kripke models for modal logic, will recognize the pair (U;R) as a so-called *world system*, where U is a set of possible worlds and R is an accessibility relation on that set of possible worlds. This means that expressions like "it is necessary that .." and "it is possible that .." can get a meaning in a given state or "world" v (see [Kr 59] or [Le 77], for example).

Example 1:

We define a simple database universe EXU (concerning employees and departments) for a fictitious company called Dyncons. First, the set-valued functions FE and FD introduce the (employee and department) attributes and their corresponding value sets. Then WE and WD determine the set of allowed employee tables and department tables, respectively. The function HF introduces the table names EMPL and DEP and their corresponding sets of allowed tables. Finally, EXU determines the set of allowed database states, expressing for instance that each department manager must be an employee with the proper maturity. (Our notations are defined in the Appendix.)

```
FE = { (NO      ; ℕ),                            | employee number
       (NAME    ; Chs(40)),                      | employee name
       (SAL     ; ℕ),                            | salary
       (SEX     ; {'M', 'F'}),                   | sex
       (DEPNO   ; [1 .. 99]),                    | department number
       (MAT     ; {'jun','anal','sr-an',         | specialisation/maturity
                   'mgr','prog','sr-pr'})};

FD = { (DNO     ; ℕ),                            | department number
       (NAME    ; Chs(45)),                      | department name
       (MANNO; ℕ)};                              | manager number

WE = {T | T ⊆ ∏(FE) and                          | the set of allowed employee tables
          {NO} is u.i. in T};
WD = {T | T ⊆ ∏(FD) and                          | the set of allowed department tables
          {DNO} is u.i. in T and
          {NAME} is u.i. in T};

HF = { (EMPL    ; WE),                           | employees
       (DEP     ; WD)};                          | departments

EXU = {v | v ∈ ∏(HF) and
          {(DEPNO; DNO)} connects v(EMPL) with v(DEP) and
          {(MANNO; NO)}  connects v(DEP) with
                               {t | t ∈ v(EMPL) and t(MAT) = 'mgr'} }.
```

Suppose now that the following dynamic constraints should hold:

(DC1) Existing department numbers must not expire
(although, for example, the name of a department is allowed to change).

(DC2) An employee must always remain within the same department.

(DC3) Salaries of employees must not decrease.

(DC4) Specialisation/maturity growth develops along the following lines:

$$\begin{tikzcd}
\text{jun} \arrow[r] \arrow[dr] & \text{anal} \arrow[r] & \text{sr-an} \arrow[r] & \text{mgr} \\
 & \text{prog} \arrow[r] & \text{sr-pr} \arrow[ur] &
\end{tikzcd}$$

expressing that a jun(ior) can become an anal(ist) or a prog(rammer), an analist can become a s(enio)r-an(alist), a senior-analist can become a manager, etc.

These constraints can be formally represented by the transition relation EXR on EXU, as defined below. In order to formalize the constraints (DC2), (DC3), and (DC4), we will assume that "in the course of time" an employee retains the same identity number.

$$\begin{array}{lll}
EXR = \{(v;v') \mid (v;v') \in EXU \times EXU \text{ and} & & \\
\quad v(DEP)\rceil\{DNO\} \subseteq v'(DEP)\rceil\{DNO\} \text{ and} & & (DC1) \\
\quad \forall t \in v(EMPL): \forall t' \in v'(EMPL): & & \\
\quad [\text{if } t(NO) = t'(NO) & & \\
\quad\ \text{then } (t(DEPNO) = t'(DEPNO) \text{ and} & & (DC2) \\
\qquad\quad t(SAL) \leq t'(SAL) \text{ and} & & (DC3) \\
\qquad\quad \text{if } t(MAT) \neq t'(MAT) & & (DC4) \\
\qquad\quad \text{then } (t(MAT); t'(MAT)) \in \{ & ('jun'\ ;'anal'), & \\
 & ('anal'\ ;'sr\text{-}an'), & \\
 & ('sr\text{-}an';'mgr'), & \\
 & ('jun'\ ;'prog'), & \\
 & ('prog';'sr\text{-}pr'), & \\
 & ('sr\text{-}pr';'mgr')\}\ )]\} &
\end{array}$$

☐ Example 1.

Usually, the transition from each state $v \in U$ to v itself should be allowed. Such a "trivial" transition might arise when we try to delete something, e.g. an employee tuple, that does not appear to occur in v. In that case, the state remains the same and the "transition" at hand happens to be (v;v). In other words, usually we want such a transition relation R to be *reflexive on* U, i.e., $\forall v \in U$: $(v;v) \in R$.
Note that the transition relation EXR is reflexive on the database universe EXU.

We note that it is sometimes inconvenient to treat dynamic constraints as "ironclad rules". After all, the situation might arise that in the end a certain transaction appeared to be erroneous or unintentional, yet cannot be undone because of the dynamic constraints. If for example department numbers (or order numbers) that are already present must not expire at any time, then the erroneous addition of a department (or an order) cannot be undone. We say in this case that the transition is *irreversible*; if it can be undone, the transition is called *reversible*:

Definition 2:
If (x;y) is any ordered pair and R is a set of ordered pairs, then:
(a) (x;y) is **reversible in** R $\Leftrightarrow$ $(y;x) \in \text{Tcl}(R)$;
(b) (x;y) is **irreversible in** R $\Leftrightarrow$ $(y;x) \notin \text{Tcl}(R)$.

If R contains only reversible elements, then R itself is also called reversible:

Definition 3:
If R is a set of ordered pairs, then:
R is **reversible** $\Leftrightarrow$ $\forall p \in R$: p is reversible in R.

We note that the transition relation EXR on EXU from Example 1 is not reversible, due to the dynamic constraint (DC1).
In many cases of non-reversible transition relations, a dynamic constraint is not used as a "law" but rather as a warning ("Do you *really* want this?") or as a condition pertaining to special authorization required to carry out the modification. In these cases this means that these dynamic constraints cannot be used as genuine transition invariants.

2. Some special classes of dynamic constraints

In this section we discuss some general forms of dynamic constraints, each introduced by some real world situation. Each form will also be illustrated by the database universe EXU or the transition relation EXR from Example 1.

Suppose we have a database skeleton g, a database universe U over g, $E \in dom(g)$, $S \subseteq g(E)$, $B \subseteq g(E)$, $C \subseteq g(E)$, $a \in g(E)$, and $b \in g(E)$, where the a-values for E are numbers. We will express each form of constraint in terms of an arbitrary transition $(v;v')$ within U.

(F1) *Cumulativity of tuples*

Some data should not be changed or deleted anymore once they are in the database, e.g. historical data. This dynamic constraint belongs to the following general class:
The E-table is *cumulative*, i.e., no E-data will be deleted or modified. In brief, no E-tuples must expire:

$v(E) \subseteq v'(E)$.

It is interesting to note that this dynamic constraint can also be regarded as a connection requirement concerning two different "points in time":

id(g(E)) connects v(E) with v′(E).

This would be quite a severe demand for E = DEP in EXU, implying that a department (number) must not expire, that the corresponding department name must not be altered, and also that the manager of that department must not be replaced. In fact the insertion of completely new departments is the only allowed modification of the department-table.
If the E-table is intended for recording historical data or logging activities for example, then this severe demand of cumulativity can be realistic, however.

(F2) *Cumulativity of all primary attribute value combinations*

Let us now suppose that a department number should not expire and the corresponding department name may not change in our company Dyncons from Example 1, but that the manager can be replaced (and new departments can also be added). This dynamic constraint is a special case of the following:

No single combination of values of *primary* attributes - i.e. attributes belonging to a minimal key - may expire in the E-table. When we denote the set of all primary attributes of the table index E in the database universe U by Prim(E,U), we can write this dynamic constraint as follows:

id(Prim(E,U)) connects v(E) with v′(E).

For E = DEP in EXU this means:
id({DNO,NAME}) connects v(DEP) with v′(DEP).

In other words, a department number should not expire and the corresponding department name may not change; however, the manager can be replaced (and new departments can also be added).
Note that the requirement (F1) implies the requirement (F2). In other words, (F2) is weaker than (F1).

(F3) <u>*Cumulativity of attribute value combinations*</u>
Sometimes, entities such as orders or payments should not be deleted anymore once they are recorded in the database - which indicates cumulativity of object identifications - and, moreover, some of their attribute values may not be changed either, e.g. creation date. This dynamic constraint belongs to the following class:
The E-table is *cumulative with respect to* S, that is, no S-values are allowed to expire:

$v(E) \upharpoonright S \subseteq v'(E) \upharpoonright S.$

Again, this can be rephrased as a connection requirement:
id(S) connects v(E) with v′(E).

In our order example above, S will consist of order number and creation date. Another example of a dynamic constraint of this form is the requirement (DC1) from Example 1, where E = DEP and S = {DNO}.
If S contains only primary attributes, e.g., if S is a minimal key of E in U, then S ⊆ Prim(E,U); in that case the requirement (F2) implies the requirement (F3).

(F4) <u>*Transition dependency*</u>

Sometimes entities such as orders or payments may be deleted once they are in the database, but the initial values of some of their attributes may not be changed anymore, e.g. creation date. This dynamic constraint is of the following general form:
For each B-value remaining in the E-table, the corresponding C-values must remain constant:

$\forall t \in v(E): \forall t' \in v'(E)$: if $t \lceil B = t' \lceil B$ then $t \lceil C = t' \lceil C$.

In our order example above, B will consist of the order number and C of the creation date. The requirement (DC2) from Example 1 is also a special case of this form. (Take U = EXU, E = EMPL, B = {NO} and C = {DEPNO}.)
If B is a key of E in U and $S = B \cup C$ in (F3), then that requirement (F3) implies (F4), i.e., then (F4) is weaker than (F3).
We will treat this class of dynamic constraints in more detail in Section 3.

(F5) *Non-decreasing attribute values*
The values of "cumulatively counting" attributes (such as *Total number of treated patients* of a department) should typically be non-decreasing. In general:
The a-value of each E-tuple in the new state must be greater than or equal to the a-value of each E-tuple with the same S-value in the old state:

$\forall t \in v(E): \forall t' \in v'(E)$: if $t \lceil S = t' \lceil S$ then $t(a) \leq t'(a)$.

Here, S is usually a key of E in U, e.g., as in requirement (DC3) in Example 1.
If B = S and C = { a } in (F4), then that requirement (F4) implies the requirement (F5).

(F6) *Non-decreasing number of tuples*
Suppose that the number of courses that our faculty offers is not allowed to decrease (although any course can be replaced by an other one), then we might have a dynamic constraint of the form below.
The number of E-tuples is not allowed to decrease:

$|v(E)| \leq |v'(E)|$.

For E = DEP in EXU this would mean that departments cannot expire all of a sudden, although they can apparently be replaced by other departments.
If S in (F3) is a key of E in U, then the requirement (F3) implies the requirement (F6), i.e., then (F6) is weaker than (F3).

(F7) *Constraints on initial values*

A constraint such as the requirement that each *new* invoice must have the status "open" is a dynamic constraint, with the following general form:
For each *initial* value for the key S of the E-table, the tuple concerned has to satisfy a certain requirement φ (in our case the requirement that the status is "open"). In other words, the S-values in the E-table of the new state v′ for which the tuples do not satisfy φ must already have occurred in the E-table of the old state v. This dynamic constraint on the admissible initial values can therefore be formulated as a connection requirement:

id(S) connects $\{ t' \in v'(E) \mid \neg \varphi(t') \}$ with v(E).

Our invoice-example above would result in something like:

id({INV-NO}) connects $\{ t' \in v'(INV) \mid t'(STATUS) \neq$ "open" $\}$ with v(INV).

We will also illustrate this class of dynamic constraints (concerning initial values) with a concrete instance based on Example 1. Suppose that we require that no new employee can immediately start as a department manager. Here,
E = EMPL, S = {NO}, and $\varphi(t') = (t'(NO) \notin \{a(MANNO) \mid a \in v'(DEP)\})$. This results in the following dynamic constraint:

id({NO}) connects $\{ t' \in v'(EMPL) \mid t'(NO) \in \{a(MANNO) \mid a \in v'(DEP)\} \}$ with v(EMPL).

(F8) *Constraints on final values*

The requirement that an invoice can only be deleted when the status is "ready" or "exit" is also a dynamic constraint, with the following general form:
A value for the key S of the E-table can only disappear when the tuple concerned satisfies a certain requirement φ. In other words, the S-values in the E-table of the old state v for which the tuples do not satisfy φ must still occur in the E-table of the new

state v'. This dynamic constraint on the final values can therefore also be formulated as a connection requirement:

id(S) connects $\{ t \in v(E) \mid \neg \varphi(t) \}$ with $v'(E)$.

Our invoice-example above would result in something like:

id({INV-NO}) connects $\{ t \in v(INV) \mid t(STATUS) \notin \{"ready", "exit"\} \}$
with $v'(INV)$.

We will also illustrate this class of dynamic constraints (concerning final values) with a concrete instance based on Example 1. Suppose that we require that a department can only disappear when it has no employees anymore. Here,
E = DEP, S = {DNO}, and $\varphi(t) = (t(DNO) \notin \{y(DEPNO) \mid y \in v(EMPL)\})$. This results in the following dynamic constraint:

id({DNO}) connects $\{ t \in v(DEP) \mid t(DNO) \in \{y(DEPNO) \mid y \in v(EMPL)\} \}$ with $v'(DEP)$.

Note that this dynamic constraint formally rules out the possibility of a proper cascading delete here (considered as one atomic transaction).

(F9) *Life cycles*

Sometimes, tuples contain some kind of "status" attribute for which only certain status transitions are allowed during their "life time", thus enforcing certain "life cycle" constraints. A classical example of such an attribute is *marital status*.
Generally, for a "status" attribute b (with value set M), proper status transitions are only allowed within a given $L \subseteq M \times M$ (for old and new tuples with the same value for a key S of the E-table):

$\{ (t(b);t'(b)) \mid (t;t') \in v(E) \times v'(E) \text{ and } t\lceil S = t'\lceil S \text{ and } t(b) \neq t'(b)\} \subseteq L$.

An example of a dynamic constraint of this form is the requirement (DC4) from Example 1, where b = MAT, E = EMPL, S = {NO}, and

M = {'jun','anal','sr-an','mgr','prog','sr-pr'} and

L = { ('jun';'anal'), ('anal' ;'sr-an'), ('sr-an';'mgr'),
('jun';'prog'), ('prog';'sr-pr'), ('sr-pr';'mgr')}

(F10) <u>*Changing life cycles*</u>

Life cycle *constraints* may be subject to change as well. E.g., our company in Example 1 may decide to add the possibility of the career step "prog → anal". If life cycles are occasionally subject to change then it is better to add a table containing the currently allowed (proper) status transitions in our database universe (i.e., we mean: better than to adapt our transition relation over and over again). In our example, that table would look like:

FROM	TO
jun	anal
anal	sr-an
sr-an	mgr
jun	prog
prog	sr-pr
sr-pr	mgr

Generally, we would extend our database skeleton g with an ordered pair, say (ST; {FROM, TO}), while the value set for the attributes FROM and TO is M (or perhaps something of the form Chs(n) if the status set M as such might be subject to change as well). This results in the following dynamic constraint allowing changing life cycles:

$\{ (t(b); t'(b)) \mid (t;t') \in v(E) \times v'(E) \text{ and } t\lceil S = t'\lceil S \text{ and } t(b) \neq t'(b)\} \subseteq$
$\{ (y(FROM); y(TO)) \mid y \in v(ST)\}.$

Note that we used v(ST) in stead of v′(ST) here. This means that status transitions have to obey the status transition conditions in the "old" state v (which is relevant in the rare situation that status transitions and status transition conditions are updated in one and the same "atomic" transaction).

Even if our life cycles are not subject to change, one might argue that this solution is more elegant than the one used in Example 1 anyway.

Note that for v = v′, the constraint forms (F1), (F2), (F3), (F6), (F7), and (F8) are always satisfied. Hence, these constraint forms can not hinder the reflexivity of any transition relation in which they appear. Also if B in (F4) and S in (F5), in (F9), and in (F10) are keys of E in U then those constraint forms are satisfied for v = v′ as well.

We notice that sometimes one of the minimal keys of E in U plays a special role, such as S in the dynamic constraints in (F3), (F5), (F7), (F8), (F9), and (F10). The intuition behind this is that in the course of time the values for that special key continue to correspond one-to-one to the "real world" objects they intend to represent. Such a special minimal key is sometimes called a *primary key*. The other minimal keys are called *alternate keys* in this regard. This topic is discussed in Chapter 3 of [Da 86], for example. Although in the literature the distinction between primary keys and alternate keys is usually made on the basis of other (often vague) criteria, the explicit role in dynamic constraints as mentioned above is in our view the most significant and concrete argument for such a distinction.

We also noticed in this section that the concept of a connection requirement is useful not only for the formulation of static constraints, but for the formulation of dynamic constraints as well. In those cases, the connecting attribute transformation is obviously an identity function.

3. Transition dependency and constant dependency

For the special class of dynamic constraints encountered in (F4) of Section 2, we shall introduce special names and notations. In the case below we will call C *transition dependent on* B *at* (T;T′), by analogy to momentary dependency.

Definition 4:
If A, B, and C are sets and T and T′ are tables over A, then:
$B \rightsquigarrow C$ **at** (T;T′) $\Leftrightarrow$ $\forall t \in T$: $\forall t' \in T'$: if $t \lceil B = t' \lceil B$ then $t \lceil C = t' \lceil C$.

The requirement (F4) in Section 2 can now be written as: $B \rightsquigarrow C$ at (v(E);v′(E)).

The next five lemmas describe some basic properties of transition dependency. We note that the properties in Lemma 1 constitute the analogon to the Armstrong axioms for momentary dependencies; see [Ar 74].

Lemma 1:
If A is a set and T and T' are tables over A, and $B \subseteq A$ and $C \subseteq A$ and $D \subseteq A$, then:
(a) if $C \subseteq B$ then $B \rightsquigarrow C$ at $(T;T')$;
(b) if $B \rightsquigarrow C$ at $(T;T')$ and $C \rightsquigarrow D$ at $(T;T')$ then $B \rightsquigarrow D$ at $(T;T')$;
(c) $B \rightsquigarrow C$ at $(T;T') \Leftrightarrow \forall c \in C$: $B \rightsquigarrow \{ c \}$ at $(T;T')$.

Lemma 1 can be proven by simply applying Definition 4. The following lemma can be proven directly from Lemma 1, i.e., without reverting to the actual definition of transition dependency.

Lemma 2:
If A is a set, T and T' are tables over A, $B \subseteq A$, $C \subseteq A$, $D \subseteq A$, and $E \subseteq A$, then:
(a) $B \rightsquigarrow B$ at $(T;T')$;
(b) if $B \rightsquigarrow C$ at $(T;T')$ and $B \subseteq D$ then $D \rightsquigarrow C$ at $(T;T')$;
(c) if $B \rightsquigarrow C$ at $(T;T')$ and $D \subseteq C$ then $B \rightsquigarrow D$ at $(T;T')$;
(d) if $B \rightsquigarrow C$ at $(T;T')$ and $D \rightsquigarrow E$ at $(T;T')$ then $B \cup D \rightsquigarrow C \cup E$ at $(T;T')$.

The following lemma treats some special cases (concerning the empty set), and follows directly from Definition 4.

Lemma 3:
If A is a set and T and T' are tables over A and $B \subseteq A$ and $C \subseteq A$, then:
(a) $\emptyset \rightsquigarrow C$ at $(T;T') \Leftrightarrow T = \emptyset$ or $T' = \emptyset$ or $|T \lceil\!\lceil C \cup T' \lceil\!\lceil C| \leq 1$;
(b) $B \rightsquigarrow \emptyset$ at $(T;T')$;
(c) $B \rightsquigarrow C$ at $(\emptyset;T')$;
(d) $B \rightsquigarrow C$ at $(T;\emptyset)$.

Lemma 4 relates transition dependency to momentary dependency (and to itself in (b)) and follows directly from Definition 4 as well.

Lemma 4:
If A is a set and T and T' are tables over A, and $B \subseteq A$ and $C \subseteq A$, then:
(a) $B \rightarrow C$ in $T \cup T'$ $\Leftrightarrow$ $B \rightsquigarrow C$ at $(T;T')$ and $B \rightarrow C$ in T and $B \rightarrow C$ in T';

(b) B ↝ C at (T;T′) ⇔ B ↝ C at (T′;T);
(c) B → C in T ⇔ B ↝ C at (T;T).

The proofs of the foregoing lemmas are left to the reader. Although these lemmas all have simple proofs, they are useful enough to be stated explicitly.
To illustrate part (a) of Lemma 4, we note that because of the uniqueness condition in the definition of WE in Example 1, the dynamic constraint (DC2) is equivalent to the *dynamic* constraint that {NO} → {DEPNO} in v(EMPL) ∪ v′(EMPL).
The following lemma shows how transition dependency interferes with some well-known table operations.

Lemma 5:
If A is a set and T and T′ are tables over A, and B ⊆ A and C ⊆ A, then:
(a) if B ↝ C at (T;T′) and X ⊆ T and X′ ⊆ T′, then B ↝ C at (X;X′);
(b) if B ↝ C at (T;T′), then B → C in (T ⋈ (T′⟦B)) ∪ (T′ ⋈ (T⟦B));
(c) if B is u.i. in T′, then:
(B ↝ C at (T;T′) and T⟦B ⊆ T′⟦B) ⇔ T⟦(B∪C) ⊆ T′⟦(B∪C).

Proof:
(a) This follows directly from Definition 4.
(b) Let X = (T ⋈ T′⟦B) and X′ = (T′ ⋈ T⟦B); now X ⊆ T and X′ ⊆ T′, so
B ↝ C at (T;T′) implies B ↝ C at (X;X′) according to part (a).
We want to prove that B → C in X and B → C in X′,
after which we can apply Lemma 4(a) to conclude that B → C in X ∪ X′.
Now, let x ∈ X and y ∈ X and x⌈B = y⌈B;
then x ∈ T and (∃x′ ∈ T′: x⌈B = x′⌈B) and y ∈ T;
hence, B ↝ C at (T;T′) implies that x⌈C = x′⌈C and y⌈C = x′⌈C
(since y⌈B = x⌈B = x′⌈B);
so, x⌈C = y⌈C. Thus, B → C in X.
Similarly, we can prove that B → C in X′.
(c) ⇒: T⟦B ⊆ T′⟦B, so ∀t ∈ T: ∃t′ ∈ T′: t⌈B = t′⌈B;
therefore, t⌈C = t′⌈C (since B ↝ C at (T;T′));
hence, t⌈(B∪C) = t′⌈(B∪C).
Thus, ∀t ∈ T: ∃t′ ∈ T′: t⌈(B∪C) = t′⌈(B∪C),
i.e., T⟦(B∪C) ⊆ T′⟦(B∪C).

$\Leftarrow$: $T \| B \subseteq T' \| B$ follows directly from $T \|(B \cup C) \subseteq T' \|(B \cup C)$.
Now, let $t \in T$ and $t' \in T'$ and $t \lceil B = t' \lceil B$;
we have to prove that $t \lceil C = t' \lceil C$.
$T \|(B \cup C) \subseteq T' \|(B \cup C)$, thus $\exists t'' \in T'$: $t \lceil (B \cup C) = t'' \lceil (B \cup C)$ since $t \in T$.
Now, $t' \lceil B = t \lceil B = t'' \lceil B$, and B is u.i. in T′, so $t' = t''$;
hence, $t \lceil C = t'' \lceil C = t' \lceil C$, which we had to prove.

□

We would like to define the concept of transition dependency also (and particularly) at the level of transition relations. We will call C *constantly dependent on* B *in* E *at* R iff C is transition dependent on B at (v(E);v′(E)) for each pair (v;v′) in R:

Definition 5:
If g is a set function, U is a database universe over g, $E \in dom(g)$, B and C are sets, and $R \subseteq U \times U$, then:
$B \rightarrow C$ **in** E **at** R $\Leftrightarrow$ $\forall(v;v') \in R$: $B \rightarrow C$ at (v(E);v′(E)).

The following five lemmas describe some basic properties of constant dependency; they are easily derived from the first five lemmas. We note that the properties in Lemma 6 again constitute the analogon to the Armstrong axioms; see [Ar 74].

Lemma 6:
If g is a set function, U is a database universe over g, $E \in dom(g)$, $B \subseteq g(E)$, $C \subseteq g(E)$, $D \subseteq g(E)$, and $R \subseteq U \times U$, then:
(a) if $C \subseteq B$ then $B \rightarrow C$ in E at R;
(b) if $B \rightarrow C$ in E at R and $C \rightarrow D$ in E at R then $B \rightarrow D$ in E at R;
(c) $B \rightarrow C$ in E at R $\Leftrightarrow$ $\forall c \in C$: $B \rightarrow \{ c \}$ in E at R.

The following lemma can be proven directly from Lemma 6, i.e., without reverting to the actual *definition* of constant dependency.

Lemma 7:
If g is a set function, U is a database universe over g, $E \in dom(g)$, $A \subseteq g(E)$, $B \subseteq g(E)$, $C \subseteq g(E)$, $D \subseteq g(E)$, and $R \subseteq U \times U$, then:
(a) $B \rightarrow B$ in E at R;

(b) if $B \rightsquigarrow C$ in E at R and $B \subseteq D$ then $D \rightsquigarrow C$ in E at R;
(c) if $B \rightsquigarrow C$ in E at R and $D \subseteq C$ then $B \rightsquigarrow D$ in E at R;
(d) if $B \rightsquigarrow C$ in E at R and $D \rightsquigarrow A$ in E at R then $B \cup D \rightsquigarrow C \cup A$ in E at R.

The following lemma treats some special cases (concerning the empty set).

Lemma 8:
If g is a set function, U is a database universe over g, $E \in \mathrm{dom}(g)$, $B \subseteq g(E)$, $C \subseteq g(E)$, and $R \subseteq U \times U$, then:
(a) $\varnothing \rightsquigarrow C$ in E at R
   $\Leftrightarrow \forall(v;v') \in R$: ($v(E) = \varnothing$ or $v'(E) = \varnothing$ or $|v(E) \upharpoonright C \cup v'(E) \upharpoonright C| \leq 1$);
(b) $B \rightsquigarrow \varnothing$ in E at R;
(c) $B \rightsquigarrow C$ in E at $\varnothing$.

Lemma 9 tries to relate constant dependency to permanent dependency (and to itself in (b)).

Lemma 9:
If g is a set function, U is a database universe over g, $E \in \mathrm{dom}(g)$, $B \subseteq g(E)$, $C \subseteq g(E)$, and $R \subseteq U \times U$, then:
(a) if $B \rightarrow C$ in E of U then:
   $B \rightsquigarrow C$ in E at R $\Leftrightarrow \forall(v;v') \in R$: $B \rightarrow C$ in $v(E) \cup v'(E)$;
(b) $B \rightsquigarrow C$ in E at R $\Leftrightarrow B \rightsquigarrow C$ in E at R^{-1}, where R^{-1} denotes the inverse of R.

The following lemma shows how constant dependency is related to cumulativity.

Lemma 10:
If g is a set function, U is a database universe over g, $E \in \mathrm{dom}(g)$, $B \subseteq g(E)$, $C \subseteq g(E)$, B is a key of E in U, and $R \subseteq U \times U$, then:
($B \rightsquigarrow C$ in E at R and $\forall(v;v') \in R$: $v(E) \upharpoonright B \subseteq v'(E) \upharpoonright B$)
$\Leftrightarrow \forall(v;v') \in R$: $v(E) \upharpoonright (B \cup C) \subseteq v'(E) \upharpoonright (B \cup C)$.

A straightforward yet important observation is that for reflexive transition relations constant dependency implies permanent dependency!

Theorem 1:
If g is a set function, U is a database universe over g, $E \in \mathrm{dom}(g)$, $B \subseteq g(E)$, $C \subseteq g(E)$, and $R \subseteq$

$U \times U$, then:
if $B \twoheadrightarrow C$ in E at R and R is reflexive on U then $B \rightarrow C$ in E of U.

Proof:
We have to prove that $\forall v \in U$: $B \rightarrow C$ in v(E);
$B \twoheadrightarrow C$ in E at R
$\Leftrightarrow \forall (v;v') \in R$: $B \twoheadrightarrow C$ at (v(E);v′(E)) (by Definition 5)
$\Rightarrow \forall v \in U$: $B \twoheadrightarrow C$ at (v(E);v(E)) (since R is reflexive on U)
$\Leftrightarrow \forall v \in U$: $B \rightarrow C$ in v(E) (by Lemma 4(c))
□

The class of transition and constant dependencies constitutes a special subclass of the more general class of dynamic constraints, and is closely related to (but different from) various other classes of "dynamic dependencies" proposed in the literature, see for example [Vi 87] and [Wij 95]. For instance, the notion of a dynamic functional dependency of Wijsen in [Wij 95] can be compared with the statement in the lefthand side of Lemma 4(a). Hence, as we can see from the righthand side of Lemma 4(a), his dynamic functional dependency is not equivalent to our transition dependency but is in fact a combination of static and dynamic dependencies.

In order to give an impression of the possible (subtle) forms of dynamic constraints appearing in the context of a database universe that is already complex in its own right, Chapter 5 of [Br 95] contains a nontrivial example of a transition relation. This example, which appeared earlier in [Br 89], also includes various "change prohibition rules".

Conclusions

We introduced a general formal theory of dynamic constraints in order to be able to represent all kinds of "real world" constraints as they can arise in a managerial context. Dynamic constraints induce a transition relation on a state space. The state space in combination with the transition relation can be considered as a Kripke model, representing a set of possible worlds and an accessibility relation on that set of possible worlds, respectively. In practice, such a transition relation will usually be a reflexive relation.
We also discussed the status of dynamic constraints since it is sometimes inconvenient to treat dynamic constraints as "ironclad rules". After all, the situation might arise that a certain

transaction appeared to be erroneous or unintentional, while it cannot be undone because of the dynamic constraints. In this case we call the transition irreversible; if it can be undone, the transition is called reversible. In many cases of non-reversible transition relations, a dynamic constraint is not used as a "law" but rather as a warning or as a condition where special authorization is required to carry out the modification.

We also reconsidered the notions of primary and alternate keys in a dynamic context. We distinguished (and exemplified) several useful classes of dynamic constraints, e.g., cumulativity, constraints on initial and on final values, life cycles, yes, even changing life cycles, and transition and constant dependencies. We also established several useful properties of these dependencies.

All in all, this paper contributed to both the theory and application of dynamic constraints, among others by establishing a general theoretical foundation for solutions to practical problems, even in subtle and complex situations.

References

[Ar 74] W.W. Armstrong: *Dependency structures of data base relationships.* Proceedings IFIP congress, North Holland, Amsterdam, 1974, pp. 580-583

[Br 89] E.O. de Brock: *Foundations of semantic databases.* (In Dutch.) Academic Service, Schoonhoven, 1989

[Br 95] E.O. de Brock: *Foundations of semantic databases.* Prentice Hall International Series in Computer Science, London, 1995

[Da 86] C.J. Date: *Relational database: Selected writings.* Addison-Wesley, Reading (Mass.), 1986

[Da 95] C.J. Date: *An introduction to database systems.* Addison-Wesley, Reading (Mass.), 1995

[Je 94] C.S. Jensen et al.: *A consensus glossary of temporal database concepts.* ACM SIGMOD Record, vol. 23(1), 1994, pp. 52-64

[Kl 93] N. Kline: *An update of the temporal database bibliography.* ACM SIGMOD Record, vol. 22(4), 1993, pp. 66-80

[Kr 59] S.A. Kripke: *A completeness theorem in modal logic.* The Journal of Symbolic Logic, vol. 24, 1959, pp. 1-14

[Le 77] E.J. Lemmon: *An introduction to modal logic.* Basil Blackwell, Oxford, 1977

[Na 89] S. Navathe and R. Ahmed: *A temporal relational model and a query language.* Information Sciences, vol. 49, 1989, pp. 147-175

[Vi 87] V. Vianu: *Dynamic functional dependencies and database aging.* Journal of the ACM, vol. 34(1), 1987, pp. 28-59

[Wij 95]J. Wijsen: *Extending dependency theory for temporal databases.* Dissertation, University of Leuven, 1995

Appendix: Basic notions

In this section we establish the definitions of the basic (database) notions as we use them in this paper (cf. [Br 95]). By a table over a set A we mean a set of functions over A, i.e., functions with A as their domain:

If A is a set, then:
T is a **table over** A ⇔ T is a set and ∀t ∈ T: t is a function over A.

An element of T is called a *tuple* and an element of A is called an *attribute* of T.
Since every table is a (special) set, every concept defined for sets also applies to tables. Thus, for example, the notions of union and intersection of two tables make sense.
A database skeleton (or database schema) can be considered as a set-valued function, assigning to each table name its set of attributes. As a frame of reference we present the simple but also well-known and widely used example in the database literature concerning suppliers, parts and shipments (cf. [Da 95]). The suppliers/parts/shipments-example has the following database skeleton, which we will call g1:

g1 = {	(S ; {S#, SNAME, STATUS, CITY}),	suppliers
	(P ; {P#, PNAME, COLOR, WEIGHT, CITY}),	parts
	(SP ; {S#, P#, QTY}) }	shipments

Here S# stands for supplier number, P# for part number, and QTY for quantity.

We define the concept of a database state (or briefly DB state) over g for any set-valued function g:

If g is a set-valued function, then:
v is a **DB state over** g ⇔ v is a function over dom(g) and
∀E ∈ dom(g): v(E) is a table over g(E).

Since every database state is a function, every concept defined for functions also applies to database states. Thus we can speak about the domain and the range of a database state, for instance. In our example above, dom(v) = {S, P, SP} for every DB state v over g1.
The set of admissible states (to be determined by the organization in question) is some set of

database states over g1. We call such a set a database universe (or briefly DB universe) over g1. In general:

If g is a set-valued function, then:
U is a **DB universe over** g ⇔ U is a set of DB states over g.

Example 1 in Section 1 contains the specification of a database universe called EXU.
If U is a DB universe over g, then we call g *the DB skeleton* (or "database schema") *of* U, an element E of dom(g) a *table index* (or "table name") *of* U, g(E) *the heading of* E *in* U, and an element of g(E) an *attribute* (or "attribute name") *of* E *in* U. We call an element of U a *DB state consistent with* U.
Since every DB universe is a (special) set, each concept defined for sets applies to DB universes as well.

The *restriction* of a tuple t to an attribute set B is denoted by t ⌈ B and
the *projection* of a table T on B is denoted by T ⌈⌈ B:
t ⌈ B = { (a;w) | (a;w) ∈ t and a ∈ B }
T ⌈⌈ B = { t ⌈ B | t ∈ T }

We denote *function composition* by f   g (f *after* g). The *identity function* on a set A is denoted by id(A): id(A) = { (x;x) | x ∈ A}.
We call the functions f and g *joinable* iff f and g match on dom(f)   dom(g), i.e., iff
∀a ∈ dom(f)   dom(g): f(a) = g(a). So, f and g are joinable iff f ∪ g is a function.

The *natural join* of the tables T and T′ is denoted by T ⋈ T′:
T ⋈ T′ = {t ∪ t′ | t ∈ T and t ∈ T′ and t and t′ are joinable }.

We define the familiar notion of functional dependency on the "incidental" level of tables (where we will call it *momentary dependency*) as well as on the "structural" level of database universes (where we will call it *permanent dependency*):

If A, B, and C are sets and T is a table over A, then:
B → C **in** T ⇔ ∀t ∈ T: ∀t′ ∈ T: if t ⌈ B = t′ ⌈ B then t ⌈ C = t′ ⌈ C.

If g is a set function, U is a DB universe over g, E ∈ dom(g), B ⊆ g(E), and C ⊆ g(E), then: B

→ C **in** E **of** U ⇔ ∀v ∈ U: B → C in v(E).

Unique identification is a special case of *momentary dependency* (namely if C = A) and, similarly, any "key" property is a special case of *permanent dependency*:

If A and B are sets and T is a table over A, then:
B **is uniquely identifying** (or **u.i.**) **in** T ⇔ ∀t ∈ T: ∀t′ ∈ T: if t⌈B = t′⌈B then t = t′.

If g is a set function, U is a DB universe over g, and E ∈ dom(g), then:
(a) B is a **key of** E **in** U ⇔ ∀v ∈ U: B is u.i. in v(E).
(b) B is a **minimal key of** E **in** U ⇔ B is a key of E in U and
∀B′   B: B′ is not a key of E in U.

The following notion constitutes a generalization of the notion of *referential integrity*. Let h be a function which maps a set B of "referencing" attributes of a table T onto a set B′ of corresponding "referenced" attributes of a table T′. (Thus, the "attribute transformation" h indicates which attributes in B correspond to which attributes in B′.) We say that h *connects* T *with* T′ iff all B-values in T also occur as B′-values in T′. Formally:

If T is a table over A, T′ is a table over A′, and h is a function with dom(h) ⊆ A and rng(h) ⊆ A′, then:
h **connects** T **with** T′ ⇔ { t⌈dom(h) | t ∈ T } ⊆ { t′   h | t′ ∈ T′}.

Often, h = id(B) for some attribute set B. The connection requirement then reduces to:
T ⟦ B ⊆ T′ ⟦ B.

In our examples, we will often use the following set notations. The set of natural numbers (including 0) is denoted by ℕ. For each n ∈ ℕ, Chs(n) denotes the set of all strings of at most n characters. If m and n are integers, then [m .. n] denotes the set of all integers between m and n (m and n included).

For each set-valued function F, ∏(F) denotes the *generalized product* of F. Formally:
∏(F) = { f | f is a function over dom(F) and ∀x ∈ dom(f): f(x) ∈ F(x) }.