# MODELING DYNAMICS OF DATABASES WITH RELATIONAL DISCRETE EVENT SYSTEMS AND MODELS

Alexander Tuzhilin
Zvi M. Kedem

Department of Information, Operations, and Management Sciences
Leonard N. Stern School of Business, New York University
44 West 4th Street, New York, NY 10012

# MODELING DYNAMICS OF DATABASES WITH RELATIONAL DISCRETE EVENT SYSTEMS AND MODELS

by

**Alexander Tuzhilin**
Leonard N. Stern School of Business
New York University
40 West 4th Street
New York, New York 10003

and

**Zvi M. Kedem**
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012-1185

**March 1991**

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

**Working Paper Series**

STERN IS-91-5

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-05

# Modeling Dynamics of Databases with Relational Discrete Event Systems and Models*

Alexander Tuzhilin†
Zvi M. Kedem‡

**Abstract**

Behavior of relational databases is studied within the framework of *Relational Discrete Event Systems (RDESes) and Models (RDEMs).* Three behavior specification methods based on production systems, recurrence equations, and Petri nets are defined and their expressive powers are compared. Production system RDEM is extended to support non-determinism, and various deterministic and non-deterministic production system interpreters are introduced and formally compared in terms of their expressive power. It is shown that the *parallel deterministic* interpreter has more expressive power than other interpreters including an OPS5-like interpreter. Since it is also parallel, this makes the parallel deterministic interpreter a very attractive interpreter for production systems.

## 1 Introduction

There has been much work done on studying behavioral aspects of databases. Examples of this work include active databases [MD89], triggers [Aea76, Sto86] and alerters [BC79], database models that explicitly support behavior, like RUBIS [LNR87], SHM+ [BR84], Event Model [KM84], integration of production and database systems into expert database systems [dMS88a, SLR88, WF90, SJGP90, sig89], modelling behavior as a sequence of states [GT86, Via87], and studying behavioral specifications with transaction languages [AV].

However, there is no unifying formal framework for studying behavior which allows comparison of the proposed models in terms of their behavioral properties as relational query languages can be compared in terms of expressive power and relational completeness. Because of that, there are many different approaches for specifying behavior which yet have to be brought together in a single formal framework.

This paper contributes towards the development of such a unifying framework by introducing the concepts of *Relational Discrete Event System (RDES)* and *Relational Discrete Event Model*

*This work was partially supported by the Office of Naval Research under the contract number N00014-85-K-0046 and by the National Science Foundation under grant number CCR-89-6949. Portions of this paper appeared in the Ph.D. thesis of the first author and in the Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1989, pp. 336–346, under the title "Relational Database Behavior: Utilizing Relational Discrete Event Systems and Models."

†Information Systems Department; Stern School of Business, New York University, 40 West 4th Street, Room 624, New York, NY 10003; Internet: tuzhilin@rnd.stern.nyu.edu.

‡Department of Computer Science; Courant Institute of Mathematical Sciences; New York University; 251 Mercer Street; New York, NY 10012-1185; Internet: kedem@cs.nyu.edu.

*(RDEM)*. RDES is defined as a set of trajectories over the space of relational database states with a specified schema and is based on the notion of a Discrete Event System (DES) [VK87]. RDEM is a formalism, based on the relational data model, which describes a possibly *infinite* RDES set of trajectories in *finite* terms[1].

The concept of a trajectory or a trace was introduced in computer science some time ago. Mazurkiewicz defined a notion of a trace in [Maz77] when he studied concurrent systems and their relationship to Petri nets. Surveys of work on trace theory include [Die90, Maz88]. In databases, sequences of database states were studied in [GT86, Via87] and in the framework of finite [Ari86, Sno87, Gad88, CC87, NA88, TC90] and infinite [CI88, KT89, Tuz89, KSW90] temporal databases. It was proposed in [VK87] to study relationships between (infinite) sets of traces (DESes) and their *finite* computational representations (DEMs).

In this paper, we adopt the notions of Discrete Event Systems and Models to traces of database states (RDESes and RDEMs) and propose to use these concepts as a basis for comparison of different models describing dynamics of databases. We also do a comparison of the RDEMs based on production systems with different interpreters, recurrence equations, and Petri net based RDEMs in terms of the sets of traces they generate. We show that the three formalisms have the same expressive power: they generate the same class of trajectories of database states. Furthermore, we consider non-deterministic RDEMs generating sets of non-deterministic trajectories and compare several non-deterministic production system RDEMs in terms of their expressive power.

The significance of the RDEM/RDES approach comes from the fact that it allows comparison of any two computational methods, describing dynamics of relational databases, in terms of the expressive power, i.e. in terms of the sets of traces the two methods can generate. For example, we show that the three methods, mentioned above, generate the same class of trajectories. This means that these three methods capture an "interesting" class of database trajectories and deserve some additional studies.

The rest of the paper is organized as follows. In Section 2 we define relational discrete event systems and models. In Section 3, we define deterministic production systems, interpreters for these systems and consider alternative conflict resolution strategies. In Section 4, we define RDEMs based on recurrence equations and compare them to production systems. In Section 5, we define RDEMs based on a special type of a Petri net, called Predicate Transition Network, and study its relationship to production systems. Finally, in Section 6, we extend production system RDEMs to support non-determinism and study several non-deterministic interpreters.

## 2 Definitions of RDEM/RDES

First, we review the concepts of Discrete Event Systems (DES) and Discrete Event Models (DEM). We will follow [Ram87] and [IV87] with some modifications of their work. Then, we define Relational Discrete Event Systems and Models as a special case of Discrete Event Systems and Models.

[1] As a comparison, a finite automaton is a formalism that characterizes a, generally, infinite language it accepts in *finite* terms.

## 2.1 Overview of Discrete Event Systems and Models

Let $A$ be a, generally infinite, alphabet, and let $A^*$ and $A^\infty$ be the sets of finite and infinite strings (sequences) over $A$ respectively. A *Discrete Event System (DES) over A*, $\Sigma(A)$, is a subset of $A^* \cup A^\infty$ such that if $\alpha$ belongs to the DES then no prefix of $\alpha$ belongs to the DES, i.e. $\alpha \in \Sigma(A) \Rightarrow (\forall\beta)(\beta \text{ is prefix of } \alpha \Rightarrow \beta \notin \Sigma(A))$. Note that a Discrete Event System can be infinite in general. A *Discrete Event Model (DEM)* is a *finite* mathematical description of the possibly *infinite* set DES.

We made an assumption that $A$ is an infinite alphabet in general. This violates the assumptions of [Ram87] and [IV87] that $A$ is finite. The need for an infinite alphabet comes from the fact that we will consider relational DESes and DEMs where $A$ can be infinite.

In this work, we restrict our attention to a special case of Discrete Event Models that can be specified as follows. Let $f$ be a *computable* function from $A^*$ to $2^A$. Let S be the smallest subset of $A^*$ satisfying the following properties: 1) $\emptyset \in S$; 2) if $\alpha \in S$ and $a \in f(\alpha)$ then $\alpha a \in S$. Note that this smallest set exists since $A^*$ satisfies this property and since the intersection of two sets satisfying this property also satisfies this property. Let $\alpha \in A^* \cup A^\infty$. Denote $\alpha_i$ to be the string consisting of the first $i$ elements of $\alpha$. Define $L_* = \{\alpha \mid \alpha \in S \wedge f(\alpha) = \emptyset\}$ and $L_\infty = \{\alpha \mid \alpha \in A^\infty \wedge \alpha_i \in S \text{ for all } i\}$. In other words, $L_*$ and $L_\infty$ are the sets of finite and infinite strings generated by function $f$. Then function $f$ defines a DEM, whose DES is $L_* \cup L_\infty$. Inan and Varaiya [IV87] call the function $f$ *event function*, and Ramadge [Ram87] calls it *supervisor*. However, [IV87] and [Ram87] do not impose any restrictions on $f$. Contrary to this, we assume that $f$ is computable. In what follows, we restrict our attention only to DEMs generated by some computable function in the manner described above. We will call $f$ a *generating function* of a DEM.

A generating function maps a string over $A$ into a *set* of alternative elements of $A$ thus resulting in the set of non-deterministic strings generated from an initial element of $A$. Therefore, generating functions are *non-deterministic* in general. However, if we assume that a generating function $f$ always maps a string over $A$ into a set always consisting of a single element of $A$ then such a generating function is *deterministic*.

The next proposition states that not all sets of trajectories can be described in finite terms. Therefore, we have to concentrate only on these sets of trajectories that can be represented with a DEM.

**Proposition 1** *Let A be an alphabet. There are DESes over A that cannot be represented by any DEM.*

**Proof:** Consider a non-recursive set $S$ of strings from $A^*$. It cannot be generated by a computable function $f$ because, otherwise, $S$ has a membership test $\alpha \in S \Leftrightarrow f(\alpha) = \emptyset$. ∎

As it follows from the proof of the proposition *any* non-recursive DES consisting of finite sequences cannot be represented with a DEM.

## 2.2 Relational Discrete Event Models

In this section, we consider Relational Discrete Event Systems and Models. Let $\mathbf{R}$ be a database schema with generally infinite domains. Let $\mathbf{\Sigma(R)}$ be the set of all the database states with schema $\mathbf{R}$.

A *Relational Discrete Event System (RDES)* is a Discrete Event System with the "alphabet" $\Sigma(\mathbf{R})$. In other words, an RDES is a set of finite and infinite sequences, called *trajectories*, over all possible database states with schema $\mathbf{R}$. Intuitively, an RDES defines the class of all possible evolutions of a relational database, and a trajectory represents an element of this class, i.e. one possible evolution of the database in time.

A *Relational Discrete Event Model (RDEM)* is a Discrete Event Model of an RDES set. In other words, an RDEM is a pair ($\mathbf{R}$, $\mathbf{F}$), where $\mathbf{R}$ is a relational schema and $\mathbf{F}$ is a generating function over the alphabet $\Sigma(\mathbf{R})$.

For example, consider a Datalog program [Ull88] $P$ which is defined over a set of predicates with schema $\mathbf{R}$ and consider some extensional database predicates (EDBs) for that schema. Then all the intermediate stages in the computation of the fixpoint of $P$ with these EBDs form a trajectory. The program $P$ is an RDEM since it describes an infinite set of trajectories in finite terms. The generating function for this RDEM is specified with Datalog program $P$. The class of RDEMs defined over all possible Datalog programs will be called the *Datalog RDEM specification method.* Similarly, we define other *RDEM specification methods* in this paper based on other mechanisms for generating trajectories, such as production systems, recurrence equations, and Petri nets.

We distinguish between deterministic and non-deterministic RDEMs depending on whether or not the generating function is deterministic or non-deterministic. A *deterministic* RDEM generates a single trajectory given its initial state and the generating function, whereas a *non-deterministic* RDEM generates a set of alternative non-deterministic trajectories from the initial state. In this section and in Sections 3, 4, 5, we will restrict our attention only to the deterministic RDEMs. In Section 6, we will consider non-deterministic RDEMs.

Given two generating functions or two RDEM specification methods, we want to be able to compare them. In order to define this comparison, we need the following preliminary concepts.

Let $TR_1$ and $TR_2$ be two trajectories, and let $TR(i)$ be the $i$-th step in trajectory $TR$. We say that a trajectory $TR_1$ is *n-congruent* to trajectory $TR_2$ if $TR_1$ is a subsequence of $TR_2$ and, for all $i$, $TR_1(i)$ and $TR_1(i+1)$ are never more than $n$ steps apart in $TR_2$. This means that any two subsequent steps in $TR_1$ cannot be arbitrarily far away in $TR_2$.

**Definition 2** *Let ($\mathbf{R}$, $\mathbf{F}_1$) and ($\mathbf{R}$, $\mathbf{F}_2$) be two RDEMs with the same schema $\mathbf{R}$. We say that the RDEM ($\mathbf{R}$, $\mathbf{F}_2$)* simulates *the RDEM ($\mathbf{R}$, $\mathbf{F}_1$) if there exists a number n such that for any initial state of the database D with schema $\mathbf{R}$, the trajectory generated by $\mathbf{F}_1$ from D is n-congruent to the trajectory generated by $\mathbf{F}_2$ from D. If $n = 1$ then we say that ($\mathbf{R}$, $\mathbf{F}_2$)* exactly simulates *($\mathbf{R}$, $\mathbf{F}_1$).*

For example the Datalog program $r(x,u) \leftarrow r(x,y) \wedge r(y,z) \wedge r(z,u)$ is simulated by the following doubly negated Datalog (Datalog$^{\neg *}$) program with the inflationary semantics [AV89] (in which negations are allowed both in the head and in the body of a rule)

$$q(x,z) \leftarrow r(x,y) \wedge r(y,z) \wedge T_0$$
$$r(x,u) \leftarrow q(x,z) \wedge r(z,u) \wedge T_1$$
$$\neg T_0, T_1 \leftarrow T_0$$
$$\neg T_1, T_0 \leftarrow T_1$$

The predicate $q$ and propositions $T_0$, $T_1$ are auxiliary, i.e. they are not part of the trajectory generated by this RDEM (only predicate $r$ forms the trajectory). Notice that the Datalog$^{\neg *}$ program simulates each step of the original program in two steps. Therefore, $n = 2$.

Let $\mathcal{F}_1(\mathbf{R})$ and $\mathcal{F}_2(\mathbf{R})$ be two RDEM specification methods defined over the same schema $\mathbf{R}$.

**Definition 3** $\mathcal{F}_1(\mathbf{R})$ dominates $\mathcal{F}_2(\mathbf{R})$ *if for any generating function* $\mathbf{F_2}$ *for the specification method* $\mathcal{F}_2(\mathbf{R})$ *there is a generating function* $\mathbf{F_1}$ *for the method* $\mathcal{F}_1(\mathbf{R})$ *such that the RDEM (*$\mathbf{R}$, $\mathbf{F_1}$*) simulates the RDEM (*$\mathbf{R}$, $\mathbf{F_2}$*). Furthermore, if the simulation is always exact, we say that* $\mathcal{F}_1(\mathbf{R})$ strongly dominates $\mathcal{F}_2(\mathbf{R})$.

For example, the RDEM specification method based on negated Datalog with inflationary semantics [AV88, KP88] strongly dominates the RDEM specification method based on Datalog. As will be shown in Section 3, the RDEM specification method based on production systems [BFK86] with the parallel interpreter, that applies all operations simultaneously, dominates the production system specification method based on the selective interpreter, that selects only one operation at a time, as OPS5 interpreter does [BFK86]).

In the next several sections, we will compare various RDEM specification methods, such as different production systems, recurrence equations and Petri nets, in terms of the type of dominance introduced in Definition 3.

# 3 Deterministic Production System RDEMs

In this section, we introduce the production system RDEM (PS RDEM), an RDEM whose generating function is specified by a production system.

A production system is defined by three components [BFK86]: a working memory that determines the state of a production system at a certain time, production rules, and by an interpreter that specifies how these rules are applied and how they change the state of the working memory. In Section 3.1 we review production rules. In Section 3.2, we define two alternative interpreters and study the relationship between them. We also formally define a PS RDEM specification method. Finally, in Section 3.3, we study the conflict resolution strategies for the interpreter that applies all the rules in parallel.

There have been many alternative models of production systems proposed in the literature [WF90, SJGP90, dMS88a, SLR88]. The special issue of the SIGMOD Record [sig89] describes this research. In this paper, we simplify these models by considering only their most important features. For instance, we consider only a single insert or delete operator in the head of a rule, and do not consider "event" clauses (*when* clause of [WF90] and *on* clause of [SJGP90]). We believe that this simplification does not diminish the power of the model and, at the same time, will help us to do formal analysis of production systems.

This paper contributes to the research on production systems in three ways. First, we present a characterization of different types of interpreters (Section 3.2 and especially Section 6.2). Second, we propose two conflict resolution strategies and study the relationship between them (Section 3.3). Third, we study non-deterministic production systems ( Section 6).

Since we consider relational DEMs in the paper, we will use relational terminology describing production systems. For example, we will associate predicates with relations and working memory with database states.

### 3.1 Overview of Production Rules

Production rules are defined as follows. Terms are defined as usual in the first order logic [End72]. We allow arbitrary recursive functions in terms. Let $O$ be the INS (insert) or DEL (delete) operator having the form $< op > (R, x_1, x_2, \ldots, x_m)$, where $< op >$ is either INS or DEL, $x_i$'s are *all distinct* variables, and $R$ is the relation to be modified. Then the format of a production rule is

$$\bigwedge_{i=1}^{m} P_i(x_{i1}, \ldots, x_{ik_i}) \rightarrow O \tag{1}$$

where $P_i$ is a literal, i.e. a predicate from **R**, a relational operator ($=, \geq$, etc.), or their negations.

Safety is defined as in [Ull88, pp. 104–106] as follows. First, we define *limited* variables in the body of a rule. Any variable that appears in a positive predicate in the body of the rule is limited. Also, any variable $X$ that appears in an expression $X = a$ in the body of the rule is limited where $a$ is a constant. If a variable $Y$ is limited and there is an expression $X = Y$ in the body of the rule then $X$ is also limited. A rule is *safe* if all the variables both in the body and the head are limited. Notice that this definition implies that any variable in the head of a rule must appear in the body as well. Only safe rules will be considered in the sequel.

**Example 1** Consider a simplified airline system AIRLINE that has a fleet of planes travelling between airports. When a plane arrives at an airport, it is scheduled for landing. Once it lands, it moves to a terminal and discharges the passengers. Then it takes new passengers, their luggage, departs the terminal and is prepared for the take-off. When its turn comes, it takes off and flies to another airport. This process continues indefinitely (we disregard maintenance in this simplified model).

Assume the state of AIRLINE is defined by the following relations:

$AIRPORT(A, TRM)$: an airport $A$ has a terminal $TRM$;
$DOCK(A, TRM, P)$: a plane $P$ is docked at a terminal $TRM$ in an airport $A$;
$LQ(A, P, POS)$: a plane $P$ is in position $POS$ in the landing queue of an airport $A$;
$TQ(A, P, POS)$: a plane $P$ is in position $POS$ in the takeoff queue of an airport $A$;
$READY\_TKOFF(A, P)$: a plane $P$ is ready for the take-off at an airport $A$;
$TK\_REQ(A, P)$: a plane $P$ submits a take-off request to the control tower at an airport $A$;
$SCHED(P, TRM)$: a plane $P$ is scheduled to arrive at a terminal $TRM$;
$DEST(P, A)$: a plane $P$ is flying towards an airport $A$.
$NEXT(P, A, A')$: a plane $P$ flies from an airport $A$ to an airport $A'$.

Below are examples of rules modeling processes in the AIRLINE system.

**P1:** If an airplane is ready for the take-off, it departs from the terminal and submits the take-off request to the control tower.

**A:** $DOCK(A, TRM, P) \wedge READY\_TKOFF(A, P) \rightarrow DEL(DOCK; A, TRM, P)$
**B:** $DOCK(A, TRM, P) \wedge READY\_TKOFF(A, P) \rightarrow INS(TK\_REQ; A, P)$

**P2:** If an airplane is the first in the take-off queue then it will take off and fly to its next destination.

**A:** $TQ(A, P, POS) \wedge POS = 1 \rightarrow DEL(TQ; A, P, POS)$

**B:** $TQ(A, P, POS) \wedge POS = 1 \wedge NEXT(P, A, A') \rightarrow INS(DEST; P, A')$

∎

Production rules, as defined in this section are related to the doubly negated Datalog (Datalog$^{\neg *}$) with inflationary semantics [AV89] since each INS operation can be interpreted as an addition of a new fact to the database and each DEL operation as a removal of an old fact. The main difference between the two formalisms is that production systems support function symbols and Datalog$^{\neg *}$ does not.

### 3.2 Interpreters

An interpreter is the third component in the definition of a production system[2]. It will be described together with the *recognize-act cycle* [BFK86] of a production system. We briefly review the steps of the recognize-act cycle using the database terminology.

In the first step of the cycle, the left-hand-side (LHS) of each rule is matched against the current state of the database. This matching process corresponds to a query against the database of all the relations appearing in the LHS of a rule. The answer to the query is called an *instantiation set*, and the tuples appearing in the instantiation set are called *instantiated tuples.* Next, project the instantiated tuples on the attributes of the relation appearing in the RHS of the rule and form an operation out of each resulting tuple. The resulting set of operations is called an *operation set of the rule.*

**Example 2** Assume AIRLINE has the current state as shown in Fig. 1. Rule **P2B** in Example 1 has variables $A$, $A'$, $P$, and $POS$. The set of the instantiated tuples for that rule is shown in Fig. 2. The set of tuples projected on attributes of relation $DEST$ is shown in Fig 3. Finally, the operation set for rule **P2B** is shown in Fig. 4.

The union of all the operations over all the production rules in a program is called an *operation set* for that program. Therefore, an operation set is the set of all the operations that *can* be performed in a given cycle. We denote an operation set as $O$.

In the second step of the recognize-act cycle, conflicts between insert and delete operations in the operation set $O$ have to be resolved. In Section 3.3, we will consider two conflict resolution strategies and compare them. The following is an example of a conflict resolution strategy to be studied in Section 3.3. If operation $INS(R, a_1, \ldots, a_n)$ conflicts with operation $DEL(R, a_1, \ldots, a_n)$, and the tuple $(a_1, \ldots, a_n)$ is in the database then $DEL$ has precedence over $INS$. As a result of this, $INS(R, a_1, \ldots, a_n)$ is removed from the operation set $O$. If $(a_1, \ldots, a_n)$ is not present in the database then $INS$ has precedence over $DEL$ and $DEL(R, a_1, \ldots, a_n)$ operation is removed from $O$. As a result of conflict resolutions, we get a reduced set $O$ containing no conflicts.

In the third step of the recognize-act cycle, a subset of operations is selected for the execution. An *interpreter* does this selection. Specifically, for a set of operations $O$, an interpreter $I$ chooses a subset of $O$ and executes all the operations in this subset. Such an interpreter will be called *deterministic* because it selects the subset of $O$ deterministically. In Section 6, we will study non-deterministic interpreters that can select several non-deterministic subsets of $O$.

[2]We will define an interpreter differently from [BFK86]. In [BFK86], an interpreter is a module that supervises the execution of production rules, whereas in this paper, the interpreter is the *part* of this module that selects a set of operations for the execution. We will provide a precise definition of an interpreter later on in this section.

| A | TRM | P |
|---|---|---|
| JFK | TRM1 | AA5 |
| JFK | TRM2 | TWA3 |
| JFK | TRM5 | PAN4 |

**DOCK**

| A | P |
|---|---|
| JFK | AA5 |
| JFK | PAN4 |
| LA | TWA7 |

**READY_TKOFF**

| A | P | POS |
|---|---|---|
| JFK | TWA5 | 1 |
| JFK | PAN2 | 2 |
| SFK | UNI3 | 1 |

**TQ**

| P | A | A' |
|---|---|---|
| TWA5 | JFK | SFK |
| PAN2 | JFK | LA |
| UNI3 | SFK | TKO |

**NEXT**

Figure 1: AIRLINE database

| A | A' | P | POS |
|---|---|---|---|
| JFK | SFK | TWA5 | 1 |
| SFK | TKO | UNI3 | 1 |

Figure 2: Instantiation Set for Rule P2B

| P | A' |
|---|---|
| TWA5 | SFK |
| UNI3 | TKO |

Figure 3: Tuples Projected on Relation **DEST**

INS(DEST; TWA5,SFK)
INS(DEST; UNI3,TKO)

Figure 4: The Operation Set for Rule P2B

Finally, the operations selected by the interpreter in the third step of the cycle are executed. This completes one iteration of the recognize-act cycle.

We will consider two deterministic interpreters. The *parallel deterministic* interpreter selects all the operations in $O$, i.e. it executes all the operations in $O$ simultaneously. Note that it is unimportant in which order operations in $O$ are executed since all the conflicts in the operation set have been resolved (this means that the interpreter has the Church-Rosser property).

The *selective deterministic* interpreter selects only one operation out of the operation set $O$ based on some *selection function* $f$. In this paper, we assume that the selection function $f$ maps a rule $R$ and the values of variables in the operation set of the rule into a numeric value. We also assume that the selection function cannot produce the same value for two different arguments. Then the operation to be scheduled for the execution in the current recognize-act cycle by the selective deterministic interpreter is determined as follows. We compute the value of the selection function for all the operations in the operation set $O$ and select that operation which has the largest value. Because of the previous assumption, there is only one operation that produces this largest value. For example, assume a set of production rules is totally ordered, and assume that there is a lexicographic ordering on constants in the operation set, e.g. assume that $(R, a_1, \ldots, a_n) \geq (Q, b_1, \ldots, b_m)$ if $R > Q$ or if $R = Q$ and $(a_1, \ldots, a_n) \geq (b_1, \ldots, b_m)$. Then $f$ can be any function monotone with respect to $\geq$. In the paper, we postulated an existence of a selection function without proposing any specific one. Although the selection of a good function is an important issue, it lies outside of the scope of this paper.

The selective deterministic interpreter is similar to the OPS5 interpreter because both of them deterministically select only one element per recognize-act cycle according to some algorithm[3]. This element is an instantiated tuple for the OPS5 interpreter and a single operation for the selective deterministic interpreter. However, the two interpreters differ in the way this element is selected. In case of the selective deterministic interpreter, the selection is based only on the values of operations in the operation set and on the rule that produced this operation. OPS5 uses several "screening" tests to eliminate other candidate elements. Contrary to the selective deterministic interpreter, these screening tests are not based strictly on the values of operations in the operation set nor on the corresponding rules. For example, one of the screening tests is based on the recency value of a tuple, i.e. how recently the tuple was inserted in the database. Because of these differences, the OPS5 interpreter cannot be considered as a special case of the selective deterministic interpreter. However, because of the similarities and because of the popularity of OPS5, we will call the selective deterministic interpreter the *OPS5-like* interpreter.

After defining the structure of production system rules and interpreters that execute these rules, we are ready to define a *production system RDEM (PS RDEM).* A *PS RDEM* is a pair (**R**, **F**), where **R** is a relational schema, and the generating function **F** is defined with a set of production rules over that schema and with an interpreter. This means that, given an initial state of the database (working memory), and a generating function defined by the set of production rules, and by an interpreter, the PS RDEM generates the trajectory of database states by repeatedly applying production rules to working memory in a recognize-act cycle.

Since each interpreter gives rise to its own PS RDEM specification method (see Section 2.2 for the definition of RDEM specification methods) it is important to compare the expressive powers of

[3]Actually, OPS5 interpreter is stochastic because the last screening test in OPS5 picks one of the remaining instantiations *at random.* However, this last screening test is applied very seldomly. Therefore, we ignore it to simplify the presentation.

these methods.

**Theorem 4** *The PS RDEM specification method based on the parallel deterministic interpreter dominates the PS RDEM specification method based on the selective deterministic interpreter.*

**Proof:** Consider a program $P$ with schema $\mathbf{R}$ and the selective interpreter. Let rule $R_i$ in $P$ have the form $Q_i(\overline{x_i}) \rightarrow O_i$ for $i = 1, \ldots, k$, where $\overline{x_i}$ is a vector of free variables in $Q_i$. Let the evaluation function of the selective interpreter be $f$. Then the equivalent program $P'$ for the parallel interpreter consists of the following pseudo-rules[4]. Each rule $R_i$ becomes

$$Q_i(\overline{x_i}) \wedge (\forall \overline{y_1}) \ldots (\forall \overline{y_k}) \bigwedge_{j=1}^{k} (Q_j(\overline{y_j}) \Rightarrow f(R_i; \overline{x_i}) \geq f(R_j; \overline{y_j})) \rightarrow O_i \qquad (2)$$

This rule selects only those tuples $\overline{x_i}$ satisfying $Q_i$ for rule $R_i$ whose evaluation function is greater than the evaluation function of all other tuples satisfying the same rule and all the tuples satisfying all other rules. This means that only the tuple with the largest evaluation function over all the rules will be selected. Therefore, programs $P$ and $P'$ produce the same RDES and, hence, are equivalent.

To finish the proof, we have to show how pseudo-rules (2) of program $P'$ can be converted into "equivalent" rules of the form (1). To do this, we will iteratively remove universal quantifiers from the LHS of rules (2) one at a time[5]. Assume, we have a rule of the form $Q_i(\overline{x_i}) \wedge \neg(\exists \overline{y}) U(\overline{x_i}, \overline{y}) \rightarrow O_i$, where $U$ is a first-order formula, and $Q_i$ is a conjunction of literals as in (2). Then replace this rule with the following set of rules:

$$\begin{aligned} T_0 &\rightarrow INS(T_1); DEL(T_0) \\ T_1 &\rightarrow INS(T_2); DEL(T_1) \\ T_2 &\rightarrow INS(T_0); DEL(T_2) \end{aligned} \qquad (3)$$

$$\begin{aligned} T_1 \wedge U(\overline{x_i}, \overline{y}) &\rightarrow INS(S; \overline{x_i}) \\ T_2 \wedge S(\overline{x_i}) &\rightarrow INS(R; \overline{x_i}) \\ T_2 \wedge S(\overline{x_i}) &\rightarrow DEL(S; \overline{x_i}) \\ T_2 \wedge R(\overline{x_i}) \wedge \neg S(\overline{x_i}) &\rightarrow DEL(R; \overline{x_i}) \\ T_0 \wedge Q_i(\overline{x_i}) \wedge \neg R(\overline{x_i}) &\rightarrow O \end{aligned} \qquad (4)$$

where the flags $T_0$, $T_1$, $T_2$, and the relations $S(\mathbf{x})$, $R(\mathbf{x})$ are temporary relations. Originally, $R = S = \emptyset$, $T_1 =$ **TRUE**, and $T_0 = T_2 =$ **FALSE**.

The predicate $R(\overline{x_i})$ simulates the formula $(\exists \overline{y}) U(\overline{x_i}, \overline{y})$, and the flags $T_0$, $T_1$, $T_2$ simulate a counter modulo 3. The purpose of the flag $T_i$ is to let the rules with that flag be executed $i$ cycles after the execution of the main rule $T_0 \wedge Q_i(\overline{x_i}) \wedge \neg R(\overline{x_i}) \rightarrow O$.

$P''$ is obtained from $P'$ by replacing (2) with rules (3) – (4) and by adding $T_0$ to *every* other rule in $P'$. Because $T_0$ is added to every rule in $P''$ inherited from $P'$, these rules are executed only once in three cycles.

[4]These rules are called "pseudo" because they contain universal quantifiers and do not have the form of (1).

[5]It was shown in [Tuz89] how this technique can be applied to arbitrary "non-normanized" production system programs, i.e. the programs that have arbitrary first order formulas in the LHS of a rule.

$P''$ is obtained from $P'$ by removing a single universal quantifier. By repeating this process iteratively, all universal quantifiers can be removed from all the pseudo-rules in $P'$, and the resulting program can be converted into the program $P'''$ with the rules of the form (1).

However, trajectories generated by $P'''$ will contain trajectories of $P$ as subtrajectories. More precisely, the trajectories of $P$ will be $n$-congruent to the trajectories of $P'''$ for some $n$. ∎

For some production system programs, the PS RDEM generated by the parallel interpreter cannot be simulated by any other program and the selective interpreter as the following example shows.

**Example 3** Let **R** consist of a relation $R(X)$, and let program $P$ have two rules $R(x) \wedge x' = x + 1 \rightarrow DEL(R, x)$ and $R(x) \wedge x' = x + 1 \rightarrow INS(R, x')$. Clearly, the RDEM defined by this program and by the parallel deterministic interpreter cannot be simulated by any production system program with the selective deterministic interpreter.

∎

Theorem 4 and Example 3 show that the parallel deterministic interpreter has more expressive power than the selective deterministic interpreter.

### 3.3 Conflict Resolution Strategies for Parallel Interpreters

In this section, we consider conflict resolution strategies for parallel interpreters. This problem has been addressed before for the RDL1 interpreter [dMS88b]. The RDL1 interpreter provides a synchronization of insert and delete operations within one rule by cancelling the two conflicting operations.

In this section, we define two additional synchronization strategies and show their equivalence. The first one, $SEM$, is semantically oriented in the sense that it is utilized by the interpreter. The second strategy $SYNT$ is syntactically oriented, since it is achieved by imposing certain syntactic restrictions on production rules that guarantee no conflicts. We describe the two strategies in turn now.

Syntactic conflict resolution strategy $SYNT$ replaces each rule

$$P \rightarrow INS(R, x_1, \ldots, x_n) \tag{5}$$

in the program with the rule

$$\neg R(x_1, \ldots, x_n) \wedge P \rightarrow INS(R, x_1, \ldots, x_n) \tag{6}$$

and the rule

$$P \rightarrow DEL(R, x_1, \ldots, x_n) \tag{7}$$

with the rule

$$R(x_1, \ldots, x_n) \wedge P \rightarrow DEL(R, x_1, \ldots, x_n) \tag{8}$$

where $P$ is a usual conjunction of literals in the LHS of a rule.

Clearly, the new program does not produce any conflicts because it always checks if a tuple is in the database before deleting it and if a tuple is not in the database before inserting it.

$SYNT$ can be enforced in two ways. First, it can be the responsibility of a programmer to write rules satisfying the syntax of (6) and of (8). Alternatively, a precompiler can transform a set of rules not satisfying $SYNT$ conditions into a set of rules that do satisfy these conditions. In the first case, it is the user who provides synchronization of parallel executions. In the second case, it is the responsibility of the precompiler.

Since $SYNT$ is achieved by syntactically modifying the program, it is not a real synchronization strategy: it does not resolve actual insertion and deletion conflicts but prevents them.

Therefore, we introduce the second conflict resolution strategy $SEM$ that semantically resolves conflicts between insertions and deletions. Since it was defined in Section 3.2 already, we only briefly review it now. If the operation set $O$ of a program has two operations $INS(R, a_1, \ldots, a_n)$ and $DEL(R, a_1, \ldots, a_n)$ and the tuple $(a_1, \ldots, a_n)$ is in the database then remove $INS(R, a_1, \ldots, a_n)$ from $O$. If $(a_1, \ldots, a_n)$ is not present in the database then remove $DEL(R, a_1, \ldots, a_n)$ operation from $O$.

It is easy to see that the two conflict resolution strategies $SEM$ and $SYNT$ are equivalent in the following sense. Let $P$ be a production systems program, and $P'$ be the program obtained from $P$ by applying $SYNT$. Assume that both $P$ and $P'$ use the parallel deterministic interpreter. Then

**Proposition 5** *$P$ with the conflict resolution strategy $SEM$ and $P'$ define the same PS RDEM.*

Proposition 5 makes $SEM$ an attractive conflict resolution strategy because it establishes its equivalence to an intuitively appealing strategy $SYNT$. Therefore, we will adopt $SEM$ as the conflict resolution strategy for parallel interpreters in the sequel.

As stated before, the language RDL1 [dMS88b] provides a different type of insert and delete synchronizations. However, as will be shown in a forthcoming paper, they are "equivalent" in some sense, i.e. one synchronization strategy can always be "reduced" to another.

To summarize, in this section we considered production system RDEMs for two types of interpreters and compared their expressive powers. We also considered two types of conflict resolution strategies and showed their equivalence. In the next section, we define an RDEM based on recurrence equations and compare it with PS RDEMs.

# 4 Recurrence Equation RDEM

In this section, we define the recurrence equation RDEM and compare it to the production system RDEM with the parallel deterministic interpreter in terms of the expressive power.

Let $\mathbf{R} = (\mathbf{R_1}(A_{11}, \ldots, A_{1k_1}), \ldots, \mathbf{R_m}(A_{m1}, \ldots, A_{mk_m}))$ be a database schema and $D^{(n)} = (R_1^{(n)}, R_2^{(n)}, \ldots, R_m^{(n)})$ be the state of a database with that schema at time $n$, where $R_i^{(n)}$ is a relation instance at time $n$. Then the *recurrence equation RDEM (RE RDEM)* is defined as $D^{(n+1)} = \mathbf{F}(D^{(n)})$, where $\mathbf{F}$ is a vector-valued function defined by

$$R_i^{(n+1)}(x_{i1}, \ldots, x_{ik_i}) = \begin{cases} TRUE & \text{if } P_i(x_{i1}, \ldots, x_{ik_i}) \\ FALSE & \text{if } Q_i(x_{i1}, \ldots, x_{ik_i}) \\ R_i^{(n)}(x_{i1}, \ldots, x_{ik_i}) & \text{otherwise} \end{cases} \quad (9)$$

where $P_i, Q_i$ are quantifier-free disjunctive normal form (DNF) formulas with predicates $R_1^{(n)}, \ldots, R_m^{(n)}$, relational operators ($=, \geq$, etc.) and their negations, such that $P_i \wedge Q_i = FALSE$. Notice that, as in production systems, we allow function symbols in recurrence equations in general. Moreover, conjunctive clauses in these DNF formulas are *safe* (see definition of safety on page 6).

The equation (9) has a natural interpretation in terms of insertions and deletions in the database. If a tuple is inserted into relation $R_i$ then $R_i^{(n+1)}(x_{i1}, \ldots, x_{ik_i})$ is true, and if it is deleted then $R_i^{(n+1)}(x_{i1}, \ldots, x_{ik_i})$ is false.

**Example 4** The recurrence equation for relation $DOCK$ in the AIRLINE system from Example 1 is

$$DOCK(A,TRM,P) = \begin{cases} TRUE & \textbf{if } LQ(A,P,POS) \wedge SCHED(P,TRM) \\ & \quad \wedge AIRPORT(A,TRM) \wedge POS = 1 \\ FALSE & \textbf{if } DOCK(A,TRM,P) \wedge \\ & \quad READY\_TKOFF(A,P) \\ DOCK(A,TRM,P) & \text{otherwise} \end{cases} \quad (10)$$

It says that if a plane is the first in the landing queue and it is scheduled to go to a terminal then dock this plane at that terminal. Also, if a plane is docked at a terminal and it is ready for the take off then move that plane away from the terminal.

To simplify the notation in this example, we dropped the time index $n$ from the equation. In this simple example, both **if** conditions consist of a single clause. In general, an **if** condition consists of several clauses connected by disjunctions.

■

The next two propositions show that production systems with parallel interpreters and recurrence equations produce RDEM specification methods equivalent in terms of their expressive power.

**Theorem 6** *The production system RDEM specification method for the parallel interpreter strongly dominates the recurrence equation RDEM specification method.*

**Proof:** Let **F** be an RE RDEM defined with the recurrence equations of the form 9. Since $P_i$ and $Q_i$ in (9) are in DNF, they can be written as $P_{i1} \vee \ldots \vee P_{ik}$ and $Q_{i1} \vee \ldots \vee Q_{im}$ respectively, where $P_{ij}$ and $Q_{ij}$ are conjunctive clauses as in the LHS of 9. For each formula (9) make the following production rules:

$$P_{ij}(x_{i1}, \ldots, x_{ik_i}) \rightarrow INS(R_i, x_{i1}, \ldots, x_{ik_i}) \quad (11)$$

$$Q_{ij}(x_{i1}, \ldots, x_{ik_i}) \rightarrow DEL(R_i, x_{i1}, \ldots, x_{ik_i}) \quad (12)$$

The PS RDEM defined by these rules generates the same set of trajectories as the RE RDEM **F**.■

The inverse of this theorem also holds.

**Theorem 7** *The recurrence equation RDEM specification method strongly dominates the production system RDEM specification method for the parallel interpreter.*

**Proof:** By Proposition 5, any production system program can be converted into an equivalent $SYNT$ program having the property that LHS's of inserts and deletes do not intersect.

For each predicate $R_i$ in this equivalent program, take all the rules having the form (11) and (12) and create the following equation

$$R_i^{(n+1)}(x_{i1},\ldots,x_{ik_i}) = \begin{cases} TRUE & \text{if } \bigvee_j P_{ij} \\ FALSE & \text{if } \bigvee_m Q_{im} \\ R_i^{(n)}(x_{i1},\ldots,x_{ik_i}) & \text{otherwise} \end{cases}$$

Variables in different $P_{ij}$, that are not among $x_{i1},\ldots,x_{ik_i}$, differ from each other for various $j$'s. The same is true for different predicates $Q_{im}$. Since the production system program satisfies the $SYNT$ requirements, the proposed RE RDEM is well defined, i.e. $\vee_j(P_{ij}) \wedge \vee_m(Q_{im}) = FALSE$. It is easy to see that the two RDEMs will always generate the same set of trajectories. ∎

# 5 Petri Net RDEM

In this section, we define an RDEM based on a special type of a Petri net, called a Predicate Transition Net (PrT Net) [Gen86], or more precisely, on its modification called Production Compilation Network (PCN) [dMS88a]. A modified Predicate Transition Network (PrT Net) consists of:

1. A bipartite directed graph $(P, T, F)$, where $P$ is a set of *places*, $T$ is a set of *transitions*, and $F$ is a set of directed arcs connecting a place $p \in P$ to a transition $t \in T$ and vice versa, so that $F \subset (P \times T) \cup (T \times P)$. Each place is associated with a predicate, generally, changing over time, and the transitions represent rules describing these changes.

2. A labeling of arcs with symbolic sums $\sum_i \alpha_i < x_i >$, where $\alpha_i$ is either +1 or -1, and $< x_i >$ is a tuple of variables, such that the arity of each tuple is the arity of the predicate associated with the adjacent place. We will also treat the sum $\sum_i \alpha_i < x_i >$ as a set of individual terms $\{\alpha_i < x_i >\}$ in order to simplify the presentation.

3. A mapping of a set of transitions $T$ into the set of formulae (called *transition selectors*) that are conjunctions of relational operators. A relational operator has the form $x\theta y$, where $x$ and $y$ are variables appearing in tuples of some sums that label the edges adjacent to the transition, and $\theta$ is a relational operator $=$, or $<$ or $\leq$.

4. A marking of places with *tokens* $t = (a_1, a_2, \ldots, a_n)$ so that the place $p$ has a token $t$ if and only if for the predicate $P$ associated with this place $P(a_1, a_2, \ldots, a_n)$ is true at that time. In other words, the tokens in the place consist of all the tuples that make the predicate at this place true.

As was observed in [dMS88a], PrT Nets exhibit similarities with production systems. In fact, the following associations can be made between the two mechanisms. A place corresponds to a predicate and a transition to a rule or a group of rules. The input places for a transition correspond to the predicates in the LHS of a rule, and each output place defines a separate production rule with the predicate from the place appearing in the RHS of that rule. The transition selector corresponds to the conditions of the rule. This correspondence is presented in Fig. 5. It turns out that the

| Production systems | PrT Nets |
|---|---|
| Rule $R$ | Transition $T$ |
| Relational Predicate $P$ | Place $P$ |
| Predicate $P(x_1, \ldots, x_n)$ in the LHS of $R$ | Label $+ < x_1, \ldots, x_n >$ on the arc $(P, T)$ |
| Predicate $\neg P(x_1, \ldots, x_n)$ in the LHS of $R$ | Label $- < x_1, \ldots, x_n >$ on the arc $(P, T)$ |
| Operation $INS(P; x_1, \ldots, x_n)$ | Label $+ < x_1, \ldots, x_n >$ on the arc $(T, P)$ |
| Operation $DEL(P; x_1, \ldots, x_n)$ | Label $- < x_1, \ldots, x_n >$ on the arc $(T, P)$ |
| Condition $x\theta y$ | Transition selector $x\theta y$ |

Figure 5: Relationship Between Production Systems and PrT Nets.

relationship between the two mechanisms, as defined in this paper, is so close that it will be proven in Theorem 8 that they define the same RDEM specification methods.

As in production systems, we impose the following safety conditions on PrT nets. For a transition $T$, we take all the labels on the incoming arcs and the transition selectors and define limited variables as in Section 3.1. Then a transition $T$ is *safe* if all these variables are limited and if all the variables in all the outgoing arcs are also limited. A PrT Net is safe if all of its transitions are safe. We consider only safe PrT Nets in the sequel.

The semantics of PrT Nets is defined as follows. As in [Gen86], tokens move between places by *firing* transitions. If a transition $T$ has incoming edges from predicates $P_1$, ..., $P_n$ and $P_i$ is labeled with the sum $\sum_j^{n_i} \alpha_{ij} < x_{ij} >$, where $\alpha_{ij}$ is either +1 or -1 and $x_{ij}$ is a tuple variable, then the transition is fired if places $P_i$ have tokens $a_{ij}$ such that the following condition holds:

$$\wedge_{i=1} \wedge_{j=1}^{n_i} \alpha_{ij} P_i(a_{ij}) \tag{13}$$

Furthermore, if place $P$ has an incoming edge from transition $T$ that is labeled with $\beta < y_1, \ldots, y_m >$ then add (delete) tokens $(t_1, \ldots, t_m)$ to (from) $P$ depending on whether $\beta$ is $+1$ or $-1$ as follows. Since the transition is safe, all $y_i$'s must occur positively in some predicate $P_i$ in (13). Then take the tokens satisfying (13) and bind variables $y_i$ to the appropriate constants in (13) producing a token $t_i$. The set of tokens $(t_1, \ldots, t_m)$ is obtained by repeating this matching process for all token $a_{ij}$ satisfying the condition (13).

**Example 5** Consider the following fragment of a PrT net, as presented in Fig. 6, partially describing an AIRLINE system from Example 4. Each place in Fig. 6 is labeled with the name of the associated predicate. Each transition describes how predicates change over time. For example, the transition $T1$ says that if $LQ(A, P, POS)$, $SCHED(P, TRM)$, and $AIRPORT(A, TRM)$ are true, and also if $POS = 1$ (the transition selector for $T1$) then $DOCK(A, TRM, P)$ will be true. Note how labels on arcs, e.g. $(A, P, POS)$, $(P, TRM)$, $(A, TRM, P)$, are used to specify what tokens will be true at the next moment in the manner similar to production systems.

The transition $T1$ is fired when the places $LQ$, $SCHED$, and $AIRPORT$ have tokens satisfying the condition

$$LQ(A, P, POS) \wedge SCHED(P, TRM) \wedge AIRPORT(A, TRM) \wedge POS = 1$$

As a result of this firing, place $DOCK$ will get new tokens $(A, TRM, P)$, where $A$, $TRM$, and $P$ satisfy the previous condition.

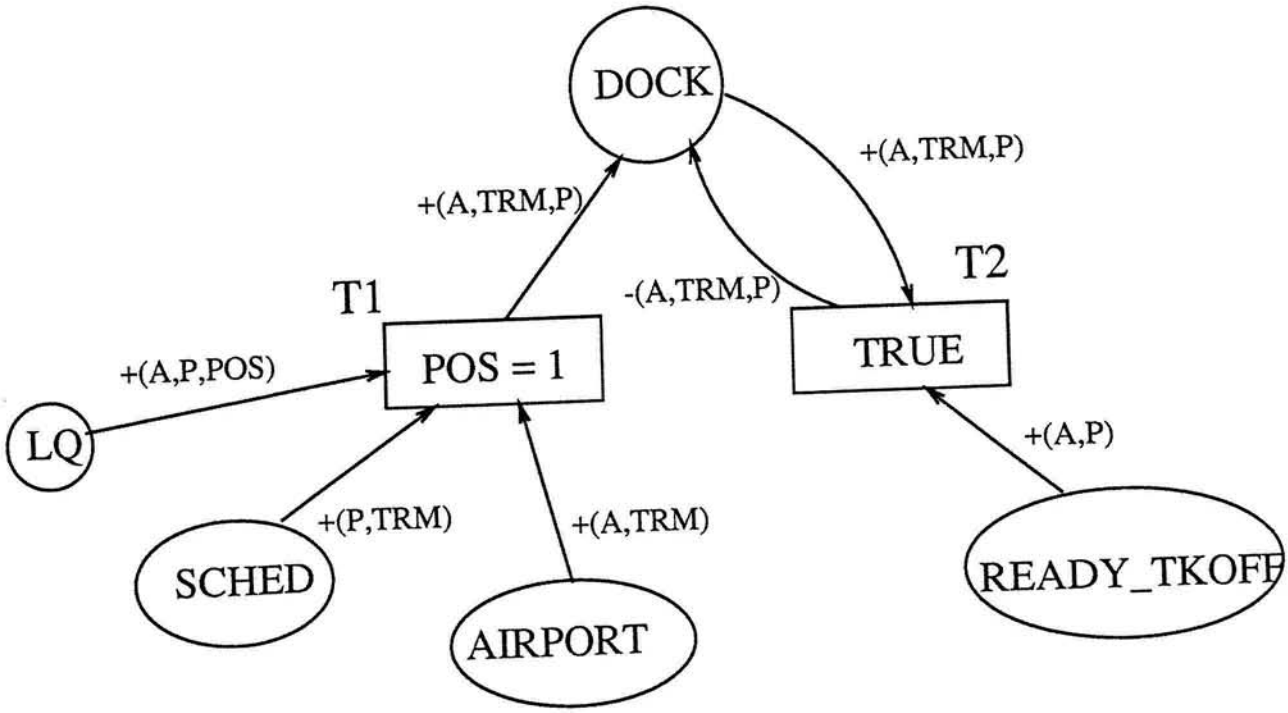


Figure 6: A Fragment of a PrT Net for the AIRPORT System.

The plus sign, $+$, in the label $+(A, TRM, P)$ on the arc between $T1$ and $DOCK$ means that the token is added to place $DOCK$. Similarly, the minus sign, $-$, in the label $-(A, TRM, P)$ between $T2$ and $DOCK$ means that the token $(A, TRM, P)$ is removed from the place $DOCK$.

In general, we allow minus signs, $-$, on the arcs from places to transitions. These minuses stand for negated predicates. However, we do not have this type of situation in this simple example.

■

We follow [dMS88a] by making the following modifications to the original definition of PrT Nets, as defined in [Gen86]. First, as proposed in [dMS88a], we assume that coefficients in sums, as defined in item 2, are either +1 or -1, as opposed to arbitrary positive coefficients in [Gen86]. Second, also following [dMS88a], we consider *conservative* nets, i.e. we assume that firing a transition $T$ will change only the states of the output places of $T$ but will not affect the input places. Third, we consider only conjunctions of relational operators in the transition selector formulas as opposed to arbitrary formulas over operators and static predicates as in [Gen86] (static predicates do not change over time). We do this for the following reasons. Since nets are conservative, we associate some places with static predicates. Therefore, there is no need to put static predicates into transitions. Also, unlike [dMS88a], we do not put negative predicates into transitions because we allow -1 coefficients in the sums of the arcs coming into transitions. Therefore, only relational operators can appear in transition selectors.

A PrT Net defines an RDEM which we call a *Petri Net RDEM*. This RDEM generates a trajectory of markings from an initial configuration of markings; each marking in this trajectory is

obtained from the previous one by firing transitions.

The following theorem establishes the relationship between production systems and PrT Nets.

**Theorem 8** *The Petri Net and the the production system RDEM specification methods are equivalent, i.e. each of them strongly dominates the other.*

**Proof:** To show that the Petri net specification method strongly dominates the production system specification method, consider a production system program $PS$. The corresponding PrT net $Pr$ is obtained out of $PS$ as follows. For each predicate $R$ in $PS$ make a place $R'$ in $P'$. For each rule $T$ in $P$, make a transition $T'$ in $P'$. If rule $T$ has an operation $INS(R; x_1, \ldots, x_n)$ in its RHS, then make an arc from the transition corresponding to that rule to the place corresponding to predicate $R$ and label that arc with $+ < x_1, \ldots, x_n >$; if the operation is $DEL(R; x_1, \ldots, x_n)$ then label that arc with $- < x_1, \ldots, x_n >$. For each predicate in the LHS of $R$, draw an arc to $T$ from the place corresponding to $R$. For each occurrence of $R(x_1, \ldots, x_n)$ in the LHS of $T$, put the label $+ < x_1, \ldots, x_n >$ on that arc if $R$ occurs positively in $T$ and the label $- < x_1, \ldots, x_n >$ if $R$ occurs negatively in $T$. For all conditions $x\theta y$ in $T$, create the transition selector with the same conditions. Clearly, the resulting PrT net $Pr$ exactly simulates $PS$.

To show that the production system specification method strongly dominates the Petri net specification method, consider a PrT net $Pr$. The production system program $PS$ that exactly simulates $Pr$ is defined as follows. Consider a transition $T$ and an arc from $T$ to some place $P$ in $Pr$. For each term $< x_1, \ldots, x_n >$ of the sum in the label on that arc, define the following production rule $R$ in $PS$. If the term $< x_1, \ldots, x_n >$ occurs positively in the sum then the RHS of $R$ is $INS(P; x_1, \ldots, x_n)$; if it occurs negatively, then the RHS of $R$ is $DEL(P; x_1, \ldots, x_n)$. The LHS of $R$ is obtained as follows. Each arc going from some place $P'$ to $T$ and each term $< x_1, \ldots, x_n >$ of the sum in the label on that arc, give rise to a conjunct $P'(x_1, \ldots, x_n)$ in the LHS of $R$. If the term occurs positively in the sum, then the predicate $P'$ also occurs positively in the LHS, and if the term occurs negatively in the sum, then the predicate also occurs negatively in the LHS. In addition, the LHS of $R$ contains conditions corresponding to the transition selector of $T$. Clearly, the resulting production system program $PS$ exactly simulates the PrT net $Pr$. ∎

It follows from Theorems 6, 7, and 8 that production system, recurrence equation and Petri net specification methods have the same expressive power. This means that the three formalisms define an important class of trajectory generating methods (RDEMs).

# 6 Non-deterministic RDEMs

We considered only deterministic RDEMs until now, i.e. RDEMs that produce a single trajectory given a generating function and an initial state of the system. In this section, we will extend this concept to non-deterministic RDEMs.

If a generating function, as defined in Section 2.1, is non-deterministic then the corresponding RDEM is also non-deterministic and can generate multiple non-deterministic trajectories. For example, consider a production system RDEM (PS RDEM), as defined in Section 3, with the interpreter that selects a single operation out of the operation set at random. Clearly, such an interpreter is non-deterministic because it can generate many non-deterministic trajectories.

In the next section, we will consider non-deterministic production system RDEMs and in Section 6.2 we compare various types of interpreters for non-deterministic PS RDEMs.

## 6.1 Non-deterministic PS RDEMs

Since non-deterministic interpreters can generate many alternative trajectories, they cannot be adequately compared with deterministic interpreters in terms of trajectories they generate: in one case, the interpreter generates multiple trajectories and in another only a single trajectory. To put deterministic and non-deterministic interpreters on an equal footing, we have to introduce non-determinism directly into production rules. In this case, both types of interpreters will generate multiple trajectories. This motivates the following definition of non-deterministic production rules.

A non-deterministic production rule has the form

$$\bigwedge_{i=1}^{m} P_i(x_{i1}, \ldots, x_{ik_i}) \rightarrow \otimes_{j=1}^{n} O_j \tag{14}$$

where $P_i$ and $O_j$ are defined as in (1) and where $\otimes$ is an EXCLUSIVE-OR operator, i.e. $O_1 \otimes O_2$ means: either apply operator $O_1$ or operator $O_2$ but not both. As in Section 3.1, we assume that rules are safe.

We consider two kinds of *rule-based non-determinism* or just *rule non-determinism.* First, *action-based non-determinism* or just *action non-determinism* is syntactically expressed with the $\otimes$ operator and defines non-deterministic alternatives among different types of operations. Specifically, it non-deterministically selects an operator $O_j$ for some $j$ in the RHS of (14). For example, the statement "if the weather is nice, I shall go for a walk, or I shall do some swimming" can be written as "weather is nice $\rightarrow$ I go for a walk $\otimes$ I do swimming." The second kind of non-determinism, called *variable-based non-determinism* or just *variable non-determinism*, expresses choices among an a priori unknown number of alternative variable assignments. To define variable-based non-determinism, we distinguish between *deterministic* and *non-deterministic* variables. We further elaborate on the variable-based non-determinism in Section 6.2 when we describe an interpreter and only explain this concept by an example at this point. Both kinds of non-determinism provide a way for the user to specify *syntactically* what kinds of non-deterministic choices he or she wants.

**Example 6** Consider the following rule for the AIRLINE example:

**P3:** If an airplane is the first in the landing queue and there are free terminals then move the plane *non-deterministically* to one of the free terminals or wait (until a better terminal becomes available).

$$LQ(A, P, POS) \wedge POS = 1 \wedge AIRPORT(A, \overline{TRM}) \wedge FREE(A, \overline{TRM})$$
$$\rightarrow \quad INS(DOCK; A, \overline{TRM}, P) \otimes WAIT$$

The variables overlined in the example are non-deterministic. Intuitively, the non-deterministic variable $\overline{TRM}$ specifies a choice among free terminals to which the airplane $P$ can move on the arrival. This is an example of a variable-based non-determinism: the set of non-deterministic alternatives is determined at the time when the rule is evaluated. The second type of non-determinism ($\otimes$) is associated with the choice between the two alternative actions: either move a plane to the selected terminal or wait (presumably for a better terminal). The set of non-deterministic alternatives is specified at the time when the program is written.

■

We introduced action and variable based non-determinism in rules to distinguish choices among several a priory known types of operators (action-based non-determinism) from choices among an unknown number of alternative variable assignments within the same operator (variable-based non-determinism).

## 6.2 Interpreters

Non-deterministic production systems execute recognize-act cycle similarly to the deterministic systems with several important differences between the two. We will describe the recognize-act cycle for the non-deterministic case while emphasizing these differences.

First, as in the deterministic case, the LHS of a rule is matched against the current state of the working memory (database) resulting in the instantiation set of the rule (see page 7).

Second, non-deterministic choices are made among the tuples in the instantiation set for non-deterministic variables in the LHS of a rule and for various choices of the $\otimes$ operator in the RHS of a rule. Specifically, partition the instantiation set of a rule based on the values of deterministic variables, i.e. all the tuples with the same values of deterministic variables go into a single partition. *Non-deterministically* select one tuple out of each partition. This selection corresponds to the variable-based non-determinism. For each selected tuple make a non-deterministic choice among alternative operators in the RHS of the rule. This choice corresponds to the action-based non-determinism. For a selected choice of alternative actions, create an operation (either insert or delete) for each instantiated tuple. The resulting set of operations forms the *operation set* for a rule. The union of all the operation sets taken over all the rules forms the operation set for the program corresponding to the non-deterministic choices already made for the rules.

**Example 7** Assume that the instantiation set for the rule **P3** from Example 6 is as shown in Fig. 7. All the tuples with the same values of deterministic variables are grouped into one set. This results in two sets of tuples as shown in Fig. 7. Non-deterministically, select one tuple in each set. An example of these non-deterministic choices is shown in Fig. 8.

■

Third, as in the deterministic case, conflicts between inserts and deletes are resolved for a specific non-deterministic choices in rules using the conflict resolution strategies described in Section 3.3.

Fourth, as in the deterministic case, a subset of operations is selected for the execution by an *interpreter*. However, a non-deterministic interpreter, unlike its deterministic counterpart, selects a *set* $\mathbf{S}(O) = \{O_1, O_2, \ldots, O_n\}$ of non-deterministic alternative subsets of $O$ (i.e. $O_i \in \mathbf{S}(O)$). After the interpreter selects the set $\mathbf{S}(O)$, all operations in the set $O_i \in \mathbf{S}(O)$ are executed simultaneously. This is done non-deterministically for all $i = 1, \ldots, n$, which means that $n$ new states are generated after the execution. Therefore, a trajectory splits into $n$ non-deterministic trajectories. This completes the recognize-act cycle.

If always $n = 1$ in the definition of the set $\mathbf{S}(O)$ then the interpreter is called *operationally deterministic* to distinguish this kind of determinism from the rule-based determinism. Otherwise, the interpreter is called *operationally non-deterministic*. If for all $i$, an interpreter selects a single operation from $O_i$ then such an interpreter is called a *sequential interpreter*; otherwise, it is called a *parallel interpreter*.

| A | P | POS | TRM |
|---|---|---|---|
| JFK | TWA5 | 1 | TRM1 |
| JFK | TWA5 | 1 | TRM3 |
| JFK | TWA5 | 1 | TRM8 |
| JFK | AA3 | 1 | TRM3 |
| JFK | AA3 | 1 | TRM5 |

Figure 7: Partitioning of the Instantiation Set for Rule P3 Based on the Non-deterministic Variable TRM

| A | P | POS | TRM |
|---|---|---|---|
| JFK | TWA5 | 1 | TRM3 |
| JFK | AA3 | 1 | TRM5 |

Figure 8: Non-deterministic Choices in the Partitioned Instantiation Set

We consider the following four types of interpreters in this paper:

- *Universal*: non-deterministically applies operations in all the subsets of $O$. This means that the universal interpreter produces $2^{|O|}$ alternative operation sets.
- *Parallel operationally deterministic*: deterministically applies all the operations in $O$, i.e. $n = 1$ and $O_1 = O$. In Section 3.2, we called this interpreter *parallel deterministic*. We added the adverb "operationally" to distinguish the non-determinism of the interpreter from the rule-based non-determinism.
- *Selective operationally non-deterministic*: each $O_i$ consists of a single operation. This interpreter produces $n$ alternative operation sets consisting of a single operation.
- *Selective operationally deterministic*: deterministically selects a single operation out of the set $O$ according to some selection function. We called this interpreter "selective deterministic" in Section 3.2.

Finally, each non-deterministically selected operation set is executed by the production system in the current recognize-act cycle. This non-deterministic choice in conjunction with non-deterministic choices in rules results in the generation of different non-deterministic alternative states of the database. Therefore, a non-deterministic PS RDEM produces a *set* of alternative trajectories from a given initial state of the database.

We considered rule-based and operational non-determinisms in this section. The user has an explicit control over the first type of non-determinism by using non-deterministic variables and operator $\otimes$. Contrary to this, the user has no control over operational non-determinism since the interpreter makes non-deterministic selections among the alternative operation sets without any interaction with the user. On the other hand, operational non-determinism constitutes a more declarative way to describe dynamics of databases since the programmer does not have to be concerned about explicit declarations of non-deterministic rules. Therefore, the two types of non-determinism are complimentary to each other.

In the next section, we study relationships between different interpreters in terms of the sets of trajectories they can generate and will show that the rule-based non-determinism is "more powerful" than the operational non-determinism for the interpreters introduced in this section.

### 6.3 Expressive Powers of Non-deterministic Interpreters

As was pointed out in Section 3.2, each deterministic interpreter gives rise to its own PS RDEM specification method. In that section, we compared the parallel deterministic and selective deterministic interpreters in terms of the PS RDEM specification methods they generate. In this section, we extend these concepts to non-deterministic interpreters and compare the parallel deterministic interpreter with the universal and selective non-deterministic interpreters.

In Section 2.2, we defined simulation and RDEM specification methods for the class of deterministic interpreters. These concepts are easily extended to the non-deterministic case. One non-deterministic PS RDEM simulates another one if there is a number $n$ such that for each initial value of EDB predicates, there is an isomorphism between non-deterministic trajectories generated by the two RDEMs such that each trajectory of the second RDEM is $n$-congruent to the isomorphic trajectory of the first RDEM. Dominance for the non-deterministic case is defined as in the deterministic case.

In the rest of this section, we compare the parallel deterministic interpreter with the universal and the selective non-deterministic interpreters. We start with the universal interpreter.

**Theorem 9** *The parallel deterministic interpreter dominates the universal interpreter.*

**Proof:** Let $P$ be a program that generates some RDES with the universal interpreter. To create the same RDES with the parallel deterministic interpreter, add *WAIT* (no-op) operator as a non-deterministic alternative via $\otimes$ operator to the RHS of each rule in $P$. ∎

**Corollary 10** *Any rule deterministic program with the universal interpreter can be simulated with a rule non-deterministic program with the parallel deterministic interpreter.*

This corollary establishes the relationship between the rule-based non-determinism of the parallel deterministic interpreter and the operational non-determinism of the universal interpreter. It says that the operational non-determinism of the universal interpreter can always be "converted" into rule non-determinism of the parallel deterministic interpreter. As was already stated before, rule-based non-determinism is more flexible than the operational non-determinism. This corollary makes the rule-based non-determinism even more "attractive" because it has more expressive power than the operational non-determinism of the universal interpreter.

The next result and Theorem 9 say that the parallel deterministic interpreter strictly dominates the universal interpreter.

**Proposition 11** *There is a program $P$ such that no program $P'$ with the universal interpreter can simulate $P$ with the parallel deterministic interpreter.*

**Proof:** Consider the program $P$

$$R(x,y) \wedge R(y,z) \rightarrow INS(R;x,z)$$

that generates the transitive closure of $R$. Assume that there is a program $P'$ and an integer $n$ such that $P$ is $n$-congruent to $P'$. Take such EDB's for $R$ that there are two consecutive states in the RDES generated by $P$ and the parallel deterministic interpreter that have more than $n$ different tuples. This means that the RDES generated by program $P'$ and the universal interpreter must have a step when the operation set of $P'$ contains more than one element. Since this is the universal interpreter, there are additional non-deterministic trajectories generated for program $P'$. This means that the RDES generated by $P'$ and by the universal interpreter does not coincide with the RDES generated by $P$ and by the parallel deterministic interpreter. ∎

Next, we want to compare the parallel deterministic and the selective non-deterministic interpreters in terms of dominance.

**Theorem 12** *The parallel deterministic interpreter dominates the selective non-deterministic interpreter.*

**Proof:** Let $P$ be a non-deterministic production system program and it has $n$ rules of the form $R_i \rightarrow \overline{O_i}$, where $R_i$ is a usual conjunction of literals, $\overline{O_i}$ is $O_{i1} \otimes \ldots \otimes O_{ik_i}$, and $O_{ij}$ is either an insert or a delete operation. We assume that the corresponding interpreter is selective non-deterministic one. Create the following program $P'$ consisting of $2^n$ rules having the form $Q_{j1} \wedge \ldots \wedge Q_{jn} \rightarrow RHS_j$, where $Q_{jm}$ is either $R_i(x_1, \ldots x_p)$ or its negation $(\forall x_1) \ldots (\forall x_p) \neg R_i(x_1, \ldots x_p)$ (therefore, there are $2^n$ formulas) and where $RHS_j$ consists of the choice $\otimes$ of $\overline{O_i}$ corresponding to those $R_i$ that occur *positively* in $j$'s rule. All the free variables in all the rules in $P'$ (and $P''$) are non-deterministic. Note that the rules in $P'$ are, generally, not in the conjunctive form because of the negated terms. However, using the same techniques as presented in the proof of Theorem 4, we can show that $P'$ can be converted to program $P''$ with conjunctive rules generating an $n$-congruent RDEM for some $n$. All the variables in $Q_{jl}$ differ from the variables in any other $Q_{jp}$ for $l \neq p$. Note that all the $2^n$ rules in $P'$ are mutually exclusive. It can be shown that program $P''$ generates the RDEM with the parallel deterministic interpreter that is $n$-congruent to the RDEM generated by $P$ with the selective non-deterministic interpreter. ∎

**Corollary 13** *Any rule deterministic program with the selective non-deterministic interpreter can be simulated with a rule non-deterministic program with the parallel deterministic interpreter.*

This corollary shows that the operational non-determinism of the selective non-deterministic interpreter can always be "converted" into the rule-based non-determinism of the parallel deterministic interpreter. The next result and Theorem 12 say that the parallel deterministic interpreter strictly dominates the selective non-deterministic interpreter.

**Proposition 14** *There is a program $P$ such that no program $P'$ with the selective non-deterministic interpreter can simulate $P$ with the parallel deterministic interpreter.*

**Proof:** Similar to the proof of Proposition 11. ∎

It follows from Theorems 9, 12 and Propositions 11, 14 that the parallel deterministic interpreter strictly dominates the universal and the selective non-deterministic interpreters. Moreover, we showed in Theorem 4 that it dominates the selective deterministic interpreter for the deterministic case. Therefore, the parallel deterministic interpreter has the following important properties.

- it is a "powerful" interpreter because it dominates other interpreters;
- operational non-determinism of the universal and selective non-deterministic interpreters can always be converted into the rule-based non-determinism of the parallel deterministic interpreter;
- it is a parallel interpreter; therefore, it provides a better performance.

These properties make the parallel deterministic interpreter an attractive interpreter for production systems.

# 7 Conclusions

In this paper, we proposed a unifying framework for modeling behavior of information intensive systems based on the concepts of Relational Discrete Event System and Model. We compared formalisms based on production systems, recurrence equations and predicate transition networks within this framework, i.e. in terms of the sets of trajectories (RDESes) they can generate. We have shown that the three formalisms are equivalent, i.e. they always generate the same sets of trajectories. This makes the class of trajectories generated by these three methods interesting and worthy of additional studies.

If we disallow function symbols in these three formalisms, then it can be easily shown that they are also equivalent to doubly negated Datalog (Datalog$^{\neg *}$) of Abiteboul and Vianu [AV89]. This observation also contributes to the fact that the three formalisms define an important class of trajectory generating methods (RDEMs).

We also extended production systems to support rule-based and operational non-determinism, and studied several deterministic and non-deterministic interpreters. We showed that the parallel operationally deterministic interpreter dominates other interpreters considered in this paper. Since this interpreter is also parallel, this makes it a very attractive type of interpreter for production systems.

We also defined a syntactic and a semantic conflict resolution strategy for production systems and showed their equivalence. Although the syntactic strategy is intuitively more appealing, the semantic strategy is more practical because it can be easily integrated with the interpreter.

# References


[Aea76] M.M. Astrahan and et al. System R: a relational approach to data. *TODS*, pages 97–137, June 1976.

[Ari86] G. Ariav. A temporally oriented data model. *TODS*, 11(4):499–527, 1986.

[AV] Serge Abiteboul and Victor Vianu. A transaction-based approach to relational database specification. *to appear in JACM.* (Technical Report CS87-102, University of California at San Diego, August 1987).

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings of PODS Symposium*, pages 240–250, 1988.

[AV89] S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *IEEE Symposium on Logic in Computer Science*, 1989.

[BC79] P. Bunemann and E. Clemons. Efficiently monitoring relational data bases. *TODS*, September 1979.

[BFK86] L. Brownston, R. Farrell, and E. Kant. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison-Wesley, 1986.

[BR84] M. L. Brodie and D. Ridjanovic. On the design and specification of database transactions. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling*, chapter 10. Springer-Verlag, 1984.

[CC87] J. Clifford and A. Croker. The historical data model (HRDM) and algebra based on lifespans. In *Proceedings of the International Conference on Data Engineering*, 1987. IEEE Computer Society.

[CI88] J. Chomicki and T. Imielinski. Temporal deductive databases and infinite objects. In *Proceedings of PODS Symposium*, pages 61–73, 1988.

[Die90] V. Diekert. *Combinatorics on Traces.* Springer-Verlag, 1990. LNCS 454.

[dMS88a] C. de Maindreville and E. Simon. Modelling non deterministic queries and updates in deductive databases. In *International Conference on Very Large Databases*, pages 395–406, 1988.

[dMS88b] C. de Maindreville and E. Simon. A production rule based approach to deductive databases. In *Proceedings of 4th International Conference on Data Engineering*, 1988.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972. New York.

[Gad88] Shashi K. Gadia. A homogeneous relational model and query languages for temporal databases. *TODS*, 13(4):418–448, 1988.

[Gen86] H. J. Genrich. Predicate/transition nets. In *Lecture Notes in Computer Science, 254*, pages 207–247. Springer-Verlag, 1986.

[GT86] S. Ginsburg and K. Tanaka. Computation-tuple sequences and object histories. *TODS*, 11(2):186–212, 1986.

[IV87] Kemal Inan and Pravin Varaiya. Finitely recursive processes. In *Discrete Event Systems: Models and Applications.* Springer-Verlag, 1987. Lecture Notes in Control and Information Sciences, 103.

[KM84] Roger King and Dennis McLeod. A unified model and methodology for conceptual database design. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling*, pages 313–327. Springer-Verlag, 1984.

[KP88] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of PODS Symposium*, pages 231–239, 1988.

[KSW90] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. In *Proceedings of PODS Symposium*, pages 392–403, 1990.

[KT89] Z. M. Kedem and A. Tuzhilin. Relational database behavior: Utilizing relational discrete event systems and models. In *Proceedings of PODS Symposium*, 1989.

[LNR87] J. Y. Lingat, P. Nobecourt, and C. Rolland. Behavior management in database applications. In *International Conference on Very Large Databases*, pages 185–196, 1987.

[Maz77] A. Mazurkiewicz. Concurrent program schemas and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[Maz88] A. Mazurkiewicz. Basic notions of trace theory. In *LInear Time, Branching Time and Partial Models in Logics and Models for Concurrency*, pages 285–363. Springer Verlag, 1988. LNCS 354.

[MD89] D. McCarthy and U. Dayal. The architecture of an active, object-oriented database system. In *Proceedings of ACM SIGMOD Conference*, 1989.

[NA88] S. B. Navathe and R. Ahmed. TSQL – a language interface for history databases. In C. Rolland, F. Bodart, and M. Leonard, editors, *Temporal Aspects in Information Systems*, pages 109–122. North-Holland, 1988.

[Ram87] Peter Ramadge. Supervisory control of discrete event systems: A survey and some new results. In *Lecture Notes in Control and Information Sciences, number 103*, pages 69–80. Springer-Verlag, 1987.

[sig89] SIGMOD Record, September 1989. Special issue on rule management and processing in expert database systems.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, cashing and views in database systems. In *Proceedings of ACM SIGMOD Conference*, pages 281 – 290, 1990.

[SLR88] T. Sellis, C.-C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proceedings of ACM SIGMOD Conference*, pages 404–412, 1988.

[Sno87] Richard Snodgrass. The temporal query language TQuel. *TODS*, 12(2):247–298, 1987.

[Sto86] M. Stonebraker. Triggers and inference in data base systems. In M. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 297–314. Springer-Verlag, 1986.

[TC90] A. Tuzhilin and J. Clifford. A temporal relational algebra as a basis for temporal relational completeness. In *International Conference on Very Large Databases*, pages 13–23, 1990.

[Tuz89] A. Tuzhilin. *Using Relational Discrete Event Systems and Models for Prediction of Future Behavior of Databases.* PhD thesis, New York University, October 1989.

[Ull88] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Via87] Victor Vianu. Dynamic functional dependencies and database aging. *JACM*, 34(1):28–59, 1987.

[VK87] P. Varaiya and A.B. Kurzhanski, editors. *Discrete Event Systems: Models and Applications*. Springer-Verlag, 1987. Lecture Notes in Control and Information Sciences, 103.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of ACM SIGMOD Conference*, pages 259 – 270, 1990.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-05