

General Purpose Schedulers for Database Systems

Marco A. Casanova and Philip A. Bernstein

Center for Research in Computing Technology, Aiken Computation Laboratory,
Harvard University, Cambridge, MA 02138, USA

Summary. A family of simple models for database systems is defined, where a system is composed of a *scheduler*, a *data manager* and several *user transactions*. The basic correctness criterion for such systems is taken to be consistency preservation. The central notion of the paper is that of a *general purpose scheduler*, a database system scheduler that is blind to the semantics of transactions and integrity assertions. Consistency preservation of a database system is shown to be precisely equivalent to a restriction on the output of a general purpose scheduler *GPS*, called *weak serializability*. That is, any database system using *GPS* will preserve consistency iff the output of *GPS* is always weakly serializable. This establishes a tight connection between database system correctness and scheduler behavior. Also, aspects of restart facilities and predeclared data accesses are discussed. Finally, several examples of schedulers correct with respect to weak serializability are presented.

1. Introduction

A database models some enterprise using both a set of data structures abstracting the relevant objects of that enterprise and a set of consistency criteria describing the allowed concrete instances of the objects. Data in the database represents a correct state of the enterprise only if it satisfies all consistency criteria. A user's program that records a change of state of the enterprise in the database must therefore modify data so as to preserve consistency. Such programs are called *transactions* [7].

If several transactions concurrently access a database, synchronization anomalies may occur. For example, consistency may be lost, a transaction may access inconsistent data or a race condition may cause the loss of the changes made by a transaction [7, 9]. Therefore, a concurrency control mechanism, or *scheduler*, must be implemented to arbitrate accesses to data. Given read and write requests submitted by transactions, the scheduler must reorder them so that database consistency is preserved.

Recent theoretical papers on database concurrency control have studied classes of read-write schedulers (e.g., [1, 2, 7, 9, 11, 16–20]) – their correctness and algorithmic complexity – and to some extent the termination properties of schedulers that produce them (e.g., [18, 20]). What has been lacking is an overall system model that accurately reflects the connection between the scheduler and the database system of which it is a component. For example, in most models “serializability” is taken to be the correctness criterion, even though correct systems that produce nonserializable schedules have been shown [12]. In one model [17], schedulers are studied with the assumption that the scheduler has complete information about every transaction that will ever be submitted, although this is unrealistic in practice. Many models do not explicitly incorporate aspects of transaction restarts (e.g., [3, 7, 16, 19, 22]), although restart facilities can be found in most database system schedulers. In this paper, we will clarify and elaborate on these points in the context of a realistic but mathematically tractable model of database schedulers.

Our goal in this paper is two-fold. First, we examine the relationship between database systems that preserve database consistency and schedulers that induce such systems. We show that a weak form of serializability for restricted forms of schedulers is precisely equivalent to consistency preservation of database systems. Second, we concentrate on practical considerations of scheduler construction. We present a schema for constructing consistency preserving schedulers, with and without a facility to restart partially executed transactions. We show that the situations when schedulers must restart transactions are closely linked to whether or not transactions predeclare the data items into which they will write. In particular, we show that if transactions do not predeclare data items they write, then restarts are necessary if a reasonable level of concurrency is to be achieved.

The overall plan of the paper goes as follows. In Sect. 2, we adopt a family of very simple models for database systems, where a system consists of user *transactions* that submit read and write requests, a *scheduler* that reorders those requests (on-line) for processing, and a *data manager* that processes the requests, responding to the user transactions. We distinguish systems with or without a restart facility and, orthogonally, systems whose transactions predeclare all data items they access, or not. We take consistency preservation as the correctness criterion for such systems.

Section 3 introduces the notion of a log as an abstraction for a sequence of read and write requests.

After this preliminary work, we introduce in Sect. 4 the central notion of the paper, *general purpose schedulers*. They are meant to integrate any database system and guarantee that the system runs correctly. Unlike [12], we assume that a general purpose scheduler knows neither the consistency criteria of the database nor the meaning of the transactions in question. Its proper operation depends only on the data items accessed by the transactions and on the ordering of the accesses. In this sense, general purpose schedulers model the concurrency control mechanisms of [3, 7, 9, 16–19, 21, 22].

We address the problem of constructing general purpose schedulers in two steps. In Sect. 5, we define a restriction on the output of a scheduler *GPS*, called *weak serializability*, first introduced in [16]. As the name implies, it is a weaker

form of the notion of serializability usually considered in the literature (see e.g., [7]). We then prove that weak serializability is equivalent to consistency preservation, in the sense that any database system using *GPS* preserves consistency iff any output of *GPS* is weakly serializable. The sufficiency of weak serializability is trivial and often quoted in the literature. Its importance stems from the fact that it provides a reasonable guideline to construct correct schedulers, whereas consistency preservation does not. The necessity of weak serializability has never been argued, as far as we know, and depends on our underlying theory. It tells us that nothing is essentially lost when we replace consistency preservation by weak serializability, within the context of general purpose schedulers. We also define another restriction on the output of a general purpose scheduler, called *conflict preserving serializability* [3], that is only a sufficient condition to guarantee consistency preservation.

Finally, in Sect. 6 we use weak serializability and conflict preserving serializability to guide the construction of schedulers guaranteeing consistency preservation. We strengthen our notion of correctness by forcing schedulers to assure that all transactions will indeed terminate properly (total correctness). For systems with predeclared information and no restarts, we exhibit practical schedulers that are totally correct. We then consider systems without predeclared information. Basically the same results are obtained if the systems have restarts, but we indicate that if no restart facility is used, the scheduler has to be so conservative that the concurrency level becomes too low. Hence, we establish an interesting tradeoff between the ability to plan ahead using predeclared information and the necessity to reorder the scheduling of operations via restarts, if no advanced information is known.

2. Database Systems

This section introduces the family of database models used throughout the paper. Our models are closely related to, but considerably more detailed than, the model in [3, 16, 17]. We first consider systems that do not permit any transaction to be reexecuted. We recognize two types of systems, depending on whether or not each transaction predeclares all data items it will access. Then, we discuss systems that allow a transaction to be reexecuted as long as it has not changed the database. Again, we recognize two types of systems, defined as above.

2.1. Database Systems Without Restarts

Let \mathcal{L} be a fixed first-order language. A *database description* consists of a pair $DB = (V, A)$, where $V = \{x_1, \dots, x_m\}$ is a set of variables of \mathcal{L} , the *database variables*, and A is a set of formulae of \mathcal{L} , the *consistency criteria*. Given a *domain* D , we associate with DB a *universe* U of states, where a *state* is an m -tuple $\bar{a} = (a_1, \dots, a_m)$ assigning a value $a_i \in D$ to each $x_i \in V$. Given an interpretation I of \mathcal{L} with domain D , we say that $\bar{a} \in U$ is *consistent* in I iff all formulae in A become valid when x_i is given a_i as value. In this case, we write $\bar{a} \models A$.

A database system *DBS*, with database $DB=(V, A)$, is modelled by a set of coroutines as in Fig. 1. The system has a set $T = \{T_1, \dots, T_n\}$ of transactions, a scheduler and a data manager. To execute, each transaction T_i sends a read request asking the values of the variables in $S(R_i) \subset V$, the readset of T_i ; it computes new values for the variables in $S(W_i) \subset V$, the writeset of T_i ; and then it sends a write request carrying the new values to be installed in the database. The scheduler accepts transaction requests as they are generated, reorders them if necessary, and passes them to the data manager. The data manager accesses the database on behalf of the transactions.

We distinguish two types of database systems. In a *type 0* system, the read request (write request) of T_i carries only the readset (writeset) of T_i . However, in a *type 1* system, the writeset is predeclared, in the sense that the read request carries both the readset and the writeset of T_i . That is, at the time the transaction requests to read, it warns the system what parts of the database it will ultimately write. Predeclared writesets will become quite important when we discuss the construction of schedulers, because they give the scheduler opportunity to improve the level of concurrency.

Assuming the format in Fig. 1, we completely specify a database system of type i ($i=0,1$) by a 5-tuple $DBS=(V, A, n, S, SCD)$, where:

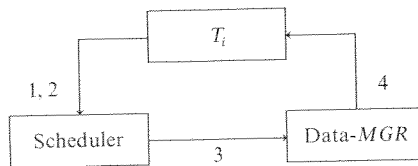
- $DB=(V, A)$ is the database description of *DBS*;
- $T=(n, S)$ describes the transaction system of *DBS*, where $S: \Sigma_n \rightarrow 2^V$, with $\Sigma_n = \{R_1, W_1, \dots, R_n, W_n\}$, specifies the readset of T_i via $S(R_i)$ and the writeset via $S(W_i)$;
- SCD is a deterministic sequential procedure called the scheduler of *DBS*; it accepts as input $(x, v) \in \Sigma_n \times S$ and returns a finite string $z \in \Sigma_n^*$; we assume that *SCD* always halts when called with parameters of the appropriate type.

The meaning of *DBS* is given by a triple $DBS^*=(D, F, I)$ where D is a domain for *DB*; F assigns to f_i ($i=1, \dots, n$) a total function $f_i^*: D^{k_i} \rightarrow D^{\ell_i}$, with $k_i = \#S(R_i)$ and $\ell_i = \#S(W_i)$, specifying the computation performed by each transaction; and I is an interpretation for \mathcal{L} with domain D .

The semantics of the programming language used in Fig. 1 follows [15], with appropriate translations for *send* and *receive* in terms of *await* and assignments. A program state of *DBS* then assigns a value to each program variable, except

Comments:

A type 0 Database System will have the following structure:



where the arrows indicate the flow of information and are labelled with the corresponding messages, and the boxes stand for the parallel blocks of the system.

Fig. 1. Type 0 database systems

database-domain, whose value is D . Hence, a program state determines a database state via the values of the variables in V . Some control information, which does not concern us here, is also carried by the program states [15]. It determines what sequences of program states can legitimately be called *computations* of *DBS* for *DBS**. Finally, we use *eval* as a function dereferencing a variable one level; for example, after the assignments $x := 1$; $y := 'x'$ *eval*(y) returns 1 and, conversely, *eval*(y) := 2 assigns 2 to x . We extend *eval* to operate on arrays in the obvious way.

We say that a computation C of *DBS* for *DBS** *terminates* when C is finite. Hence, C terminates iff each process of *DBS* either terminates or waits forever on a *receive* statement. In particular, deadlock is a form of termination.

Messages:

- (1) **read-request_{*i*}**: a message containing the following fields (with respective values)
 - symbol**: $R_i \in \Sigma_n$ (op.symbol = 'R' and tr.symbol = i)
 - variables**: a list of the variables in $S(R_i)$ ordered by index
- (2) **write-request_{*i*}**: a message containing the following fields
 - symbol**: $W_i \in \Sigma_n$ (op.symbol = 'W' and tr.symbol = i)
 - variables**: a list of the variables in $S(W_i)$ ordered by index
 - values**: a list of values for the variables in **variables**, constructed with elements of the value of **database-domain**
- (3) **Request, access-request**: messages containing the following fields
 - symbol**: an element of Σ_n
 - variables**: a list of elements of $\{x_1, \dots, x_m\}$
 - values**: a list of elements of the value of **database-domain**
- (4) **answer**: either 'ACK' or a list as **values-request**

DBS: cobegin

```

T( $i: 1 \dots n$ ): begin send read-requesti to scheduler-input;
               receive values-readi from transaction-input( $i$ );
               values.write-requesti :=  $f_i$ (values-readi);
               send write-requesti to scheduler-input;
               receive ack from transaction-input( $i$ );
               end

```

Scheduler: while true do

```

begin receive request from scheduler-input;
  STORE(request);
  X := {(symbol.request, variables.request)};
  [LI := LI || symbol.request;]
  [SI := SI ∪ X;]
  SCD(symbol.request, X; output-list);
  while output-list ≠ A do
    begin  $x := \text{car}(\text{output-list})$ ; output-list := cdr(output-list)
      if RETRIEVE( $x$ ) ≠ A
        then send RETRIEVE( $x$ )
          to data-manager-input;
      end
    end
  end

```

Data-MGR: while true do

```

begin receive access-request from data-manager-input;
  if op.symbol.access-request = 'R'
    then answer := eval(variables.access-request);
    else begin answer := 'ACK';
      eval(variables.access-request) := values.access-request;
    end
  [LO := LO || symbol.access-request;]
  send answer to transaction-input(tr.symbol.access-request);
end

```

coend

Notes:

- (a) **car**(*s*): returns the first symbol of a string *s*;
cdr(*s*): returns *s* with the first symbol deleted.
- (b) We leave undefined two auxiliary functions:
 - **STORE**(**request**): bookkeeps **request** with key **symbol** in **request-table**;
 - **RÉTRIEVE**(*x*): if *x* is the key of some entry in **request-table**, that entry is returned and deleted from **request-table**; otherwise *A* is returned.
- (c) For type 1 systems, **read-request**_{*t*} contains a new field **writeset** defined similarly to **variables** in **write-request**_{*t*}. The messages **request** and **access-request** also contain a **writeset** field just like **variables**. The scheduler is modified as follows:

```
Scheduler: while true do
  begin receive request from scheduler-input;
  STORE(request);
  X := {(symbol.request, variables, request)};
  if op.symbol.request = 'R'
    then X := X ∪ {( 'W' || tr.symbol.request, writeset-request )};
  ;
end □
```

2.2. Database Systems with a Restart Facility

In this section we extend our model to include a very restricted restart facility. We allow the scheduler to reexecute a transaction as long as the write request of the transaction has not been serviced by the data manager, that is, as long as the output of the transaction has not been installed in the database. Note that no checkpoint/rollback mechanism is needed here.

The reasons for choosing the above form of restart facility will become clear in Sect. 6. Briefly, we will show that, for systems whose transactions do not predeclare the writeset (type 0 systems), reasonable schedulers without restarts cannot be built. This problem disappears, though, when restarts are permitted.

The format of database systems with restarts is very similar to that of systems without restarts. A transaction consists again of five steps. The first four remain unchanged, but the last is modified as follows:

- the reply to the write request can either be an 'ACK' or new values for the variables in the readset. If it is an 'ACK', the transaction terminates; otherwise, execution is resumed at the third step (that is, the transaction is effectively restarted from the beginning with new values for its readset).

The scheduler and the data manager remain unchanged. All bookkeeping related to restarts is hidden in the procedure *SCD*. We also distinguish two types of systems with restarts, defined as before. Again, we completely specify a database system with restarts by a 5-tuple $DBSR = (V, A, n, S, SCD)$, whose semantics is given by a triple $DBSR^* = (D, F, I)$ as before.

2.3. Assessing the Models

The models described in Sects. 2.1 and 2.2 abstract directly a centralized database system. They can be extended in two directions to bring them closer to real systems.

First, the transactions' readsets and writesets may be described by predicates, as in [7], instead of sets of variables. Second, transactions may issue several read requests, instead of just one, but only one write request. Since neither extensions impact the results that follow, the models described need not be more detailed to convey our message to database practitioners.

However, the results in later sections fail if multiple write requests per transaction are allowed. In a centralized database system, we do not believe that this extension is significant, though, because the data a transaction modifies should be installed in the database only when the transaction terminates, for reliability reasons [13]. This is exactly equivalent to allowing only one write (to the database) per transaction just prior to termination.

As far as distributed database systems are concerned, the models described apply only when the concurrency control subsystem resides in a single node, as in [21]. In this case, the data manager abstracts several processes running on different nodes of the network. However, when the concurrency control decisions are taken by a set of processes running on different nodes, we have to consider multiple writes per transactions (see [5]). Hence, as mentioned before, our results do not apply to this case.

2.4. Correctness Criteria for Database Systems

We now define correctness criteria for database systems and their interpretations. We will often refer to criteria of this type as *database predicates*. The most basic correctness criterion, *well-formedness*, states that the scheduler must output only messages it received, which implies that the read message of a transaction is always output before the write message. We also define a system to be *total* iff all transactions terminate and all new values generated are indeed stored in the database.

We say that *DBS preserves consistency* for *DBS** iff, for any finite computation *C* of *DBS* for *DBS**, *C* maps the set of consistent database states into itself. Since it is very rare to find an application that does not require consistency preservation, this criterion is almost universally imposed. However, it does not imply that each transaction reads consistent data or that race conditions do not occur. Further correctness criteria have to be imposed, if such anomalies must be avoided. We refer the reader to [4] for a full discussion of this point.

3. Logs

It is clear from the discussion in Sect. 2 that the scheduler controls the concurrent execution of the transactions by relaying the messages it receives to the data manager in a different order. Thus, to discuss concurrency control strategies, it is convenient to introduce a special notation for streams of messages. Section 3.1 provides such notation via the concept of *log*. Section 3.2 then states a convenient result relating logs to computations.

3.1. Definition of Log

Let $DBS = (V, A, n, S, DBS)$ be a database system (with or without restart) and DBS^* be an interpretation for DBS . Recall that a message received or sent by the scheduler is of the form $M = (x, v, a)$, where $x \in \Sigma_n$, $v \subset V$ and a assigns values to the variables in v . Consider now the auxiliary variables LI , LO , SI defined in Fig. 1 (within brackets), which we assume are not modified elsewhere. In any state c of a computation C of DBS for DBS^* , the values of these variables are:

- LI_c (LO_c): a string in Σ_n^* abstracting the sequence of messages received (sent) by the scheduler up to state c . LI_c (LO_c) is called an *input (output) log* of DBS for DBS^* ;

- SI_c : a function from Σ_n to 2^v , which is the subset of S known to the scheduler up to state c . The pair (LI_c, SI_c) is called an *augmented input log* of DBS for DBS^* , and the pair (LO_c, SI_c) , an *augmented output log* of DBS for DBS^* .

We now redefine the notion of log independently of the database system, assuming first no restarts. Let $V = \{x_1, \dots, x_m\}$ be the set of database variables and consider the following sets:

- $\Sigma = \{R_p, W_p/p \in [1, \infty)\}$
- $\Sigma^* = \{\text{strings of symbols from } \Sigma\}$
- $\Gamma = \{L \in \Sigma^*/\text{no } x \in \Sigma \text{ occurs twice in } L \text{ and, if } W_p \in \Sigma \text{ occurs in } L, R_p \text{ also occurs in } L \text{ and precedes } W_p\}$
- $\Omega = \{L \in \Gamma/\text{for any } p \in [1, \infty), \text{ if } R_p \text{ occurs in } L \text{ then } W_p \text{ also occurs in } L\}$
- $\Phi = \{F: \Sigma \rightarrow 2^v, F \text{ a partial function}\}$
- $\Psi = \{F: \Sigma \rightarrow 2^v, F \text{ a total function}\}$

Strings in Σ^* are called *logs*, those in Γ are called *well-formed logs*, and those in Ω are called *complete logs*. An *augmented log* is a pair $(L, S) \in \Sigma^* \times \Phi$. In view of the database types defined in Sect. 2.1, we also introduce the sets:

- $\text{TYPE}[0] = \{(L, S) \in \Gamma \times \Phi/\text{DOMAIN}(S) = \text{elem}(L)\}$
- $\text{TYPE}[1] = \{(L, S) \in \Gamma \times \Phi/\text{DOMAIN}(S) = \text{elem}(L) \cup \{W_p \in \Sigma/R_p \in \text{elem}(L)\}\}$

and the *generic type*:

- $G = \{(L, S) \in \Gamma \times \Phi/\text{elem}(L) \subset \text{DOMAIN}(S)\}$

where $\text{elem}(L)$ denotes the set of symbols occurring in L .

Assume now systems with restarts. The definition of well-formed logs must be modified as follows:

- for output logs, Γ now reads
- $$\Gamma^0 = \{L \in \Sigma^*/\text{no } W_p \in \Sigma \text{ occurs twice and, if } W_p \in \Sigma \text{ occurs in } L, R_p \text{ also occurs in } L \text{ and all occurrences of } R_p \text{ precede } W_p\}$$
- for input logs, Γ now reads
- $$\Sigma^I = \{L \in \Sigma^*/\text{no } R_p \in \Sigma \text{ occurs twice and, if } W_p \in \Sigma \text{ occurs in } L, R_p \text{ also occurs in } L \text{ and precedes all occurrences of } W_p\}.$$

We often drop the superscript and let the context indicate which version of Γ is being referred to. Each version of Γ has a corresponding version of Ω , $\text{TYPE}[0]$, $\text{TYPE}[1]$, and G defined as before.

Finally, we define Σ_I^* , etc., by replacing Σ by $\Sigma_I = \{R_p, W_p/p \in I\}$; when $I = [l, n]$, we abbreviate Σ_I by Σ_n , and similarly for the other sets (to be consistent with Sect. 2.1).

3.2. Relating Logs and Computations

Let DBS be a database system (with or without restarts), DBS^* be an interpretation for DBS and $C = (C_0, \dots, C_j, \dots)$ be a computation of DBS for DBS^* . We first note that the value of LO in C_j records the way all accesses to the database were interleaved up to C_j . If we also know the database variables affected by these accesses, we can discover if a transaction reads the output of another. Furthermore, by using the meaning of the transactions, we can in fact recover the sequence of changes made to the initial database state (the values of the database variables in C_0). The net effect is to allow us to work with augmented logs in place of computations in later theorems. Specifically, we will be able to state consistency preservation in terms of augmented logs. To state these observations precisely, we associate a straight line program schema $P[L, S]$ with each $(L, S) \in G \cup G^0$, defined as follows:

- replace R_i in L by “ $\bar{r}_i := \bar{x}_i$ ”, where \bar{x}_i is $S(R_i)$ ordered by index;
- replace W_i in L by “ $\bar{x}'_i := f_i(\bar{r}_i)$ ”, where \bar{x}'_i is $S(W_i)$ ordered by index.

An interpretation P^* and a Herbrand interpretation P_H^* for $p[L, S]$ are defined exactly as in program schema theory [14]. That is, P^* is a pair (D, F) where D is a domain for the variables in V and F assigns a function $f_i^*: D^{k_i} \rightarrow D^{\ell_i}$ to each f_i , with $k_i = \#S(R_i)$ and $\ell_i = \#S(W_i)$. Since $p[L, S]$ has no tests, the Herbrand interpretation is unique. Each interpretation P^* induces a function $p_{P^*}[L, S]: D^m \rightarrow D^m$ such that $p_{P^*}[L, S](\bar{a}) = \bar{b}$ iff \bar{a} and \bar{b} are the initial and final values of the variables in V for a computation of $p[L, S]$ with interpretation P^* . As an abuse of language, we say that any pair $Q^* = (D, F')$ or any triple $DBS^* = (D, F', I)$, with $F \subset F'$, is also an interpretation of $p[L, S]$.

We contend that $p[L, S]$ captures the intuitive discussion in the beginning of this section, in the following sense. Let $DBS, DBS^*, C = (C_0, \dots, C_j, \dots)$ be as before. Denote by \bar{x}_{c_j} the values of the database variables of DBS in state $C_j \in C$. Then, $p_{DBS^*}[LO_{c_j}, SI_{c_j}](\bar{x}_{c_0})$ equals to \bar{x}_{c_j} .

Theorem 3.1. *Let DBS be a database system with interpretation DBS^* . Suppose that DBS is well-formed for DBS^* . Then*

$$(\forall C \in \text{comp}[DBS, DBS^*])(\forall C_j \in C)(\bar{x}_{c_j} = p_{DBS^*}[LO_{c_j}, SI_{c_j}](\bar{x}_{c_0}))$$

Sketch of Proof. Let DBS be a database system with interpretation $DBS^* = (D, F, I)$. Suppose DBS is well-formed for DBS^* . One can construct a formula P expressing that

(1) if the current values of LO and SI are L and S , then the current value of $x_j \in V$ is given by $p_{DBS^*}[L, S](\bar{x}_0)$;

(2) for any transaction T_i whose first *receive* has been executed, the value of *values-read* $_i$, \bar{v}_i , is given by $p_{DBS^*}[L, S]$, where L is the prefix of L up to, but excluding R_i ;

(3) for any transaction T_i whose second *send* has been executed, the value of *values.write-request* $_i$ is indeed given by $f_i^*(\bar{v}_i)$, where $f_i^* \in F$.

If $\bar{x} = \bar{x}_0$ is a precondition for DBS , then P can be proved to be an invariant of DBS , hence establishing our claim. \square

Using Theorem 3.1, we can restate consistency preservation in terms of logs.

Corollary 3.1. *Let $DBS = (V, A, n, S, SCD)$ be a database system with interpretation DBS^* . Suppose DBS is well-formed for DBS^* . Then DBS preserves consistency for DBS^* iff*

$$(\forall C \in \text{comp}[DBS, DBS^*])(\bar{x}_{C_0} \models A \text{ implies } p_{DBS^*}[LO_{C|c|}, SI_{C|c|}](\bar{x}_{C_0}) \models A)$$

where $C|c|$ denotes the last state of C . \square

4. General Purpose Scheduler

The rest of the paper concerns a class of schedulers, called *general purpose schedulers*, that can act as schedulers for any database system and guarantee that the system will run correctly. Our class models the kind of concurrency control mechanisms discussed in [3, 7, 9, 16–20, 22] to some extent, but not those in [12]. Further classes can be found in [11].

This section lists the basic characteristics of the class and defines it accordingly. Herbrand interpretations for database systems are also introduced to simplify the discussion. The remaining sections focus on the construction of schedulers in this class guaranteeing that any database system using them will be correct with respect to (abbreviated w.r.t.) the database predicates defined in Sect. 2.4.

4.1. Characterization of General Purpose Schedulers

For a scheduler GPS to qualify as a *type i ($i=0, 1$) general purpose scheduler* for database systems without restarts (with restarts), GPS must meet the following criteria. First, GPS cannot be specific to any particular set of transactions and database description, since it must be capable of integrating any database system. So, we require:

(1) GPS must work for any type i database system without restarts (with restarts) $DBS = (V, A, n, S, GPS)$ with any interpretation DBS^* .

Note that (1) does not rule out a complex scheduler that has an off-line component that preanalyzes DBS and DBS^* producing an on-line module tuned to the application in question. Such a scheduler would be an automated version of the approach described in [12]. We eliminate these schedulers in two phases. The next criterion requires that GPS be blind to the semantics of DBS , that is, to DBS^* and the consistency criteria of the database in question:

(2) GPS has no access to DBS^* or A .

But (2) leaves room for concurrency control mechanisms, like those described in [2, 17], that are based on a preanalysis of the syntax of DBS , as represented by V and the transaction system $TS = (n, S)$. In a distributed environment, as that of *SDD-1* [2], this preanalysis is well justified because it reduces the synchronization overhead; by studying $TS = (n, S)$ in advance, it is possible to limit the amount of information shipped across the network just to detect and avoid synchronization anomalies. However, since our model is not specific to such an environment, we

cannot support this assumption. Hence, we require GPS to be a fully on-line scheduler:

(3) the response of GPS must depend only on information contained in the messages received, as expressed by the current value of LI and SI , and on the scheduling thus far, summarized by the current value of LO .

Note that (3) implies that, for type 0 systems, GPS has no “lookahead” on S , whereas for type 1 systems GPS obtains only predeclared writesets.

Before giving a more precise characterization of general purpose schedulers, we introduce some nonstandard notation:

– $db_s_i[GPS]$ is the set of all database systems of type i , $i=0, 1$, of the form $DBS=(V, A, n, S, GPS)$, for a given scheduler GPS . We assume that the context dictates if the restart facility is being considered or not;

– $int[DBS]$ is the set of all interpretations for DBS ;

– $comp[DBS, DBS^*]$ is the set of all computations of DBS for DBS^* ;

– $trace[DBS, DBS^*]$ is the set of all traces of DBS for DBS^* .

We define a *trace* of DBS , with interpretation DBS^* , as a sequence $t \in (\Sigma^* \times \Phi \times \Sigma^* \times \Sigma^*)^*$ obtained from some computation C of DBS for DBS^* as follows:

– let c be the state where SCD is called for the i -th time in C (counting from 1) and let d be the state where SCD returns. Then $t_i=(LI_c, SI_c, LO_c, output.list_d)$.

We understand t as capturing the information contained in C that SCD sees or creates. Denote by t_j the j -th element of t and by t_j^k the k -th coordinate of t_j ($k \in [1, 4]$). Our basic definition reflects (3) above almost directly:

Definition 4.1. A scheduler GPS for type i database systems ($i=0, 1$) is a *type i general purpose scheduler* iff

$$\begin{aligned} & (\forall DBS_1 \in db_s_i[GPS])(\forall DBS_1^* \in int[DBS_1])(\forall t_1 \in trace[DBS_1, DBS_1^*]) \\ & (\forall DBS_2 \in db_s_i[GPS])(\forall DBS_2^* \in int[DBS_2])(\forall t_2 \in trace[DBS_2, DBS_2^*]) \\ & (\forall i \in [1, \min(|t_1|, |t_2|)])(\forall k \in [1, 3])(t_{1,i}^k = t_{2,i}^k \Rightarrow t_{1,i}^4 = t_{2,i}^4). \quad \square \end{aligned}$$

Note that (1) is implicit in the form of quantification used in Definition 4.1, and (2) follows from Proposition 4.1 below.

Proposition 4.1. *Let GPS be a type i general purpose scheduler.*

$$\begin{aligned} & (\forall DBS_1=(V_1, A_1, n_1, S_1, GPS) \in db_s_i[GPS])(\forall DBS_1^* \in int[DBS_1]) \\ & (\forall DBS_2=(V_2, A_2, n_2, S_2, GPS) \in db_s_i[GPS])(\forall DBS_2^* \in int[DBS_2]) \\ & (n_1 = n_2 \wedge S_1 = S_2 \Rightarrow trace[DBS_1, DBS_1^*] = trace[DBS_2, DBS_2^*]). \quad \square \end{aligned}$$

Both Definition 4.1 and Proposition 4.1 apply to systems with or without restarts.

4.2. Herbrand Interpretations

We can simplify the treatment of general purpose schedulers by defining a special domain and special interpretations patterned after Herbrand interpretations in program schema theory [14]. Given a database system $DBS=(V, A, n, S, SCD)$ a *Herbrand interpretation* for DBS is a triple $H_{DBS}=(H_D, H_F, H_I)$ such that

- H_D is the *Herbrand universe* of DBS constructed as follows: if $x_i \in V$ then “ x_i ” $\in H_D$; for each $i \in [1, n]$, $j \in [1, \ell_i]$ and t_1, \dots, t_{k_i} in H_D , “ $f_{ij}(t_1, \dots, t_{k_i})$ ” $\in H_D$, where $\ell_i = \# S(W_i)$ and $k_i = \# S(R_i)$;
- H_F assigns to each function symbol f_i ($i = 1, \dots, n$) the function over H_D , $f_i^* = (f_{i1}^*, \dots, f_{i\ell_i}^*)$, where f_{ij}^* maps $t_1, \dots, t_{k_i} \in H_D$ into the string “ $f_{ij}(t_1, \dots, t_{k_i})$ ”;
- H_I is any interpretation for \mathcal{L} with domain D_H .

Hence, H_D and H_F are defined exactly as in program schema theory, except that each transaction of DBS is not completely uninterpreted, and so, not a schema proper. Finally, note that H_D and H_F are unique.

Let $hint[DBS]$ be the set of all Herbrand interpretations for DBS and let $htrace[DBS, H_{DBS}]$ denote all traces obtained from computations of DBS for H_{DBS} whose initial state assigns “ x_i ” as value of x_i , for each $x_i \in V$.

Proposition 4.2. *Let GPS be a type i general purpose scheduler ($i=0, 1$).*

$$(\forall DBS \in dbs_i[GPS])(\forall DBS^* \in int[DBS])(\forall H_{DBS} \in hint[DBS]) \\ (trace[DBS, DBS^*] = htrace[DBS, H_{DBS}]). \quad \square$$

5. Correctness Criteria for General Purpose Schedulers

Recall that a type i general purpose scheduler GPS must work for any type i database system DBS with any interpretation DBS^* . Hence, we say that GPS is *correct w.r.t. a database predicate Q* (or, simply that GPS is Q) iff, for any database system DBS using GPS , with any interpretation DBS^* , DBS and DBS^* satisfy Q .

The question of constructing a general purpose scheduler correct w.r.t. consistency preservation immediately raises a problem: the definition of consistency preservation gives no indication on how to construct correct systems, let alone general purpose schedulers. To help solve this problem, we introduce the notation of a *log predicate P* defining a subset of $\Sigma^* \times \Phi$ and imposing restrictions on the output of a general purpose scheduler. As we will see log predicates can be devised that define guidelines for constructing general purpose schedulers and correspond exactly to consistency preservation. In the literature, log predicates are typically chosen as the fundamental notion of scheduler correctness.

We say that GPS is *correct w.r.t. a log predicate P* iff, for any database system DBS using GPS , with any interpretation DBS^* , any computation of DBS for DBS^* terminates with a final augmented output log (L, S) satisfying P . We recognize two relationships between a database predicate Q and a log predicate P . We say that P is *equivalent to (approximates) Q* iff, for any type i general purpose scheduler GPS , GPS is correct w.r.t. Q iff (if) GPS is correct w.r.t. P .

In this section we introduce two log predicates, weak serializability [16] and conflict preserving serializability [3], and prove that the former is equivalent to consistency preservation while the latter approximates it. We leave for Sect. 6 the question of using log predicates to guide the construction of schedulers w.r.t. database predicates.

Systems without restart are assumed throughout this section except in Sect. 5.3.

5.1. Weak Serializability

We say that an augmented log (L, S) is *weakly serializable* iff, for any interpretation P^* chosen for $p[L, S]$ there is an augmented serial log (L, S) – one in which reads and writes with different indexes are not interleaved – such that $p_{P^*}[L, S] = p_{P^*}[L, S]$. In intuitive terms, given any interpretation for a transaction system $T = (n, S)$, any interleaved computation of T abstracted by (L, S) produces the same changes on the database variables as some serial computation of a subset of T – one in which reads and writes from different transactions are not interleaved.

Weak serializability approximates consistency preservation almost by definition. For assume that the transactions' interpretations are such that each transaction preserves consistency. Then, each serial computation also preserves consistency, and so does each computation abstracted by a weakly serializable log.

We now state these ideas precisely.

Definition 5.1. $(L, S) \in SE$ iff $(L, S) \in G \wedge (\exists i_1, \dots, i_n \in [1, \infty))(L = R_{i_1} W_{i_1} \dots R_{i_n} W_{i_n})$. We say that $(L, S) \in SE$ is a *serial augmented log*. \square

Definition 5.2. $(L, S) \in WSR$ iff $(L, S) \in G \wedge (\exists (L', S') \in SE)$

$$(elem(L) \subseteq elem(L') \wedge S' \subseteq S \wedge p_H[L, S](\bar{x}) = p_H[L', S'](\bar{x}))$$

where H is the Herbrand interpretation of (L, S) and $\bar{x} = (\bar{x}_1, \dots, \bar{x}_m)$. We say that $(L, S) \in WSR$ is weakly serializable and (L', S') is a *weak serialization* of (L, S) . \square

The requirement that $elem(L) \subseteq elem(L')$ and $S' \subseteq S$ assures that the Herbrand interpretation of (L, S) will be a superset of the Herbrand interpretation of (L', S') . Hence, Definition 5.2 is well constructed. In the usual notion of serializability, L is taken to be a complete log, that is, $elem(L) = elem(L)$ and $S' = S$. In [4], we argue that serializability is unnecessarily strong as far as consistency preservation is concerned.

Let $int[L, S]$ be the set of all interpretations of (L, S) . Then, we can give an alternative characterization of WSR , closer to our intuitive explanation:

Proposition 5.1. $(L, S) \in WSR$ iff $(L, S) \in G \wedge (\exists (L', S') \in SE)$

$$(elem(L) \subseteq elem(L') \wedge S' \subseteq S \wedge (\forall P^* \in int[L, S])(p_{P^*}[L, S] = p_{P^*}[L', S'])). \quad \square$$

In words, (L, S) is weakly serializable if we can find a serial augmented log (L', S') such that $elem(L) \subseteq elem(L')$, $S' \subseteq S$ and $p[L, S]$ has the same input-output behavior as $p[L', S']$, for any interpretation P^* .

Example 5.1. Let $(L, S) \in G$ be denoted by $L_1[S(L_1)] \dots L_k[S(L_k)]$

(a) $(A, \phi) \in SE \cap WSR$

(b) $R_1[x] R_2[y] W_1[x] W_2[x] \in WSR$ with weak serialization

$R_1[x] W_1[x] R_2[y] W_2[x]$

(c) $R_1[x] R_2[x] W_2[x] W_1[x] \in WSR$ with weak serialization $R_1[x] W_1[x]$

Note that this log is not serializable in the strong sense, since there is no serialization containing both transactions.

(d) $R_1[x] R_2[y] W_1[y] W_2[x] \notin WSR$

(e) $R_1[x] R_2[x] W_1[x] \in WSR$ with weak serialization $R_1[x] W_1[x]$. \square

To state our first result, we revise our notion of correctness. We say that a type i general purpose scheduler is *conditionally correct* w.r.t. consistency preservation iff

$$(\forall DBS \in \text{dbs}_i[GPS])(\forall DBS^* \in \text{intc}[DBS])(DBS \text{ preserves consistency for } DBS^*)$$

where $\text{intc}[DBS]$ is the set of all interpretations $DBS^* = (D, F, I)$ of DBS such that each f_i^* induces a mapping that takes the set of consistent states into itself.

We now prove that WSR approximates consistency preservation.

Theorem 5.1. *Let GPS be a type i general purpose scheduler ($i=0, 1$). Suppose GPS is well-formed. Then, if GPS is correct w.r.t. WSR , then GPS is conditionally correct w.r.t. consistency preservation.*

Proof. Assume GPS is well-formed and correct w.r.t. WSR . Then we have:

$$\begin{aligned} & (\forall DBS \in \text{dbs}_i[GPS])(\forall DBS^* \in \text{int}[DBS])(\forall C \in \text{comp}[DBS, DBS^*]) \\ & (|C| < \infty \Rightarrow (LO_{C_{|e|}}, SI_{C_{|e|}}) \in WSR). \end{aligned} \quad (1)$$

By Corollary 3.1, we have to prove that

$$\begin{aligned} & (\forall DBS = (V, A, n, S, GPS) \in \text{dbs}_i[GPS])(\forall DBS^* \in \text{intc}[DBS]) \\ & (\forall C \in \text{comp}[DBS, DBS^*])(\bar{x}_{C_0} \models A \Rightarrow p_{DBS^*}[LO_{C_{|e|}}, SI_{C_{|e|}}](\bar{x}_{C_0}) \models A). \end{aligned} \quad (2)$$

Let $DBS = (V, A, n, S, GPS)$, $DBS^* \in \text{intc}[DBS]$ and $C \in \text{comp}[DBS, DBS^*]$. Assume $\bar{x}_{C_0} \models A$. Since DBS has no restarts, we have:

$$|C| < \infty. \quad (3)$$

Let $M = LO_{C_{|e|}}$ and $R = SI_{C_{|e|}}$. As GPS is well-formed and DBS is type i , we have:

$$(M, R) \in G. \quad (4)$$

By (1), (3), Definition 5.2 and Proposition 5.1, there is $(M', R') \in SE$ such that $\text{elem}(M') \subseteq \text{elem}(M)$, $R' \subseteq R$ and

$$(\forall P^* \in \text{int}[M, R])(p_{P^*}[M, R] = p_{P^*}[M', R']) \quad (5)$$

in particular, as $DBS^* \in \text{int}[M, S]$, we have:

$$p_{DBS^*}[M, R] = p_{DBS^*}[M', R']. \quad (6)$$

But, as each function in DBS^* induces a mapping taking the set of consistent states into itself, by construction of $p[M', R']$ and assumption on \bar{x}_{C_0} , we have:

$$p_{DBS^*}[M', R'](\bar{x}_{C_0}) \models A. \quad (7)$$

Finally, by (6) and (7), we obtain:

$$p_{DBS^*}[M, R](\bar{x}_{C_0}) \models A. \quad \square \quad (8)$$

Theorem 5.1 says that we can construct general purpose schedulers correct w.r.t. *WSR* and safely assert that they are conditionally correct w.r.t. consistency preservation. However, one might argue that there could be a general purpose scheduler correct w.r.t. consistency preservation that does not follow *WSR*. We now show that, in our theory, this is impossible.

The converse of Theorem 5.1 relies on the fact that a general purpose scheduler *GPS* is blind to the consistency criteria *A* and the interpretation *DBS** of $DBS = (V, A, n, S, GPS)$. Hence, we can assume that *A* is a special criterion to be defined below and *DBS** is a Herbrand interpretation of the system. We choose *A* to mean that a database state \bar{x} is consistent iff it can be reached from " \bar{x} " by a sequential execution of the transactions of *DBS*, possibly with repetition. We recall that " \bar{x} " = (" x_1 ", ..., " x_m "), when $V = \{x_1, \dots, x_m\}$. The interesting fact is that, in any Herbrand interpretation for *DBS* such that *A* is given the above interpretation, all transactions preserve consistency. We now state these ideas more precisely.

Given a database system $DBS = (V, A, n, S, SCD)$, with $V = \{x_1, \dots, x_m\}$, the serial Herbrand universe \bar{H}_{DBS} for *DBS* is a set of *m*-tuples of strings over the alphabet

$\{\text{"}x_i\text{"}/x_i \in V\} \cup \{\text{"}f_{ij}\text{"}/1 \leq i \leq n \wedge 1 \leq j \leq \#S(W_i)\}$ defined as follows:

- (a) " \bar{x} " = (" x_1 ", ..., " x_m ") $\in \bar{H}_{DBS}$
 (b) if $\bar{y} = (y_1, \dots, y_m) \in \bar{H}_{DBS}$, then $\bar{y}' = (y'_1, \dots, y'_m) \in \bar{H}_{DBS}$, where for some $j \in [1, n]$, for any $i \in [1, m]$,

if $x_i \notin S(W_j)$ then $y'_i = y_i$

otherwise $y'_i = \text{"}f_{ji}(y_{j_1}, \dots, y_{j_{\ell_j}})\text{"}$, where $S(R_j) = \{x_{j_1}, \dots, x_{j_{\ell_j}}\}$ (y'_i is a string).

Note that \bar{H}_{DBS} depends only on V, n , and S . Consider now a subset of \bar{H}_{DBS} :

$\bar{H}_{DBS} = \{(y_1, \dots, y_m) \in \bar{H}_{DBS} / \text{for any } i \in [1, m], \text{ for any } j \in [1, n], \text{ for any } k \in [1, \#S(W_j)], \text{"}f_{jk}\text{" occurs at most once in } y_i\}$.

By construction of \bar{H}_{DBS} , the following two lemmas hold.

Lemma 5.1. *Let $DBS = (V, A, n, S, SCD)$ be a database system and H_{DBS} be a Herbrand interpretation for *DBS*.*

$$\bar{y} \in \bar{H}_{DBS} \quad \text{iff } (\exists (L, S) \in SE)(L \in \Omega_I \wedge p_{H_{DBS}}[L, S](\bar{x}))$$

where $I = \{i \in [1, n] / \text{"}f_{ij}\text{" occurs in some } y_i\}$. \square

In what follows we assume that \mathcal{L} , the language used to write database descriptions, has an *m*-ary predicate symbol A^m , for each $m \in [1, \infty)$.

Lemma 5.2. *Let $DBS = (V, \{A^m(\bar{x})\}, n, S, SCD)$ be a database system with $m = \#V$ and let $H_{DBS} = (H_D, H_F, H_I)$ be a Herbrand interpretation for *DBS*. Suppose that $A^m_{H_I}(\bar{a}) = \text{true}$ iff $\bar{a} \in \bar{H}_{DBS}$. Then, $H_{DBS} \in \text{intc}[DBS]$. \square*

Theorem 5.2. *Let *GPS* be a type *i* general purpose scheduler ($i = 0, 1$). Suppose *GPS* is well-formed. Then, if *GPS* is conditionally correct w.r.t. consistency preservation, then *GPS* is correct w.r.t. *WSR*.*

Proof. Assume GPS is well-formed and conditionally correct w.r.t. consistency preservation. Then, by Cor. 3.1 and since every computation is always finite, GPS satisfies:

$$\begin{aligned} & (\forall DBS = (V, A, n, S, GPS) \in \text{dbs}_i[GPS]) (\forall DBS^* \in \text{intc}[DBS]) \\ & (\forall C \in \text{comp}[DBS, DBS^*]) (\bar{x}_{C_0} \models A \Rightarrow p_{DBS^*}[LO_{C_{|e|}}, SI_{C_{|e|}}](\bar{x}_{C_0}) \models A). \end{aligned} \quad (1)$$

We have to prove that GPS satisfies

$$\begin{aligned} & (\forall DBS = (V, A, n, S, GPS) \in \text{dbs}_i[GPS]) (\forall DBS^* \in \text{int}[DBS]) \\ & (\forall C \in \text{comp}[DBS, DBS^*]) (|C| < \infty \Rightarrow (LO_{C_{|e|}}, SI_{C_{|e|}}) \in WSR). \end{aligned} \quad (2)$$

By Propositions 4.1 and 4.2, (2) is equivalent to

$$\begin{aligned} & (\forall DBS = (V, \{A^m(\bar{x})\}, n, S, GPS) \in \text{dbs}_i[GPS]) (\forall C \in \text{comp}[DBS, H_{DBS}]) \\ & (\bar{x}_{C_0} = \text{``}\bar{x}\text{''} \Rightarrow (LO_{C_{|e|}}, SI_{C_{|e|}}) \in WSR), \end{aligned} \quad (3)$$

where H_{DBS} is chosen to satisfy the conditions of Lemma 1.2. Let

$$DBS = (V, \{A^m(\bar{x})\}, n, S, GPS) \in \text{dbs}_i[GPS], \quad H_{DBS} \in \text{hint}[DBS]$$

satisfying the conditions of Lemma 5.2 and $C \in \text{comp}[DBS, H_{DBS}]$ be such that $\bar{x}_{C_0} = \text{``}\bar{x}\text{''}$. Let $L = LI_{C_{|e|}}$, $R = SI_{C_{|e|}}$ and $M = LO_{C_{|e|}}$. By Lemma 5.2, $H_{DBS} \in \text{intc}[DBS]$, and since $\text{``}\bar{x}\text{''} \in \bar{H}_{DBS}$, by (1) we have:

$$p_{H_{DBS}}[M, R](\text{``}\bar{x}\text{''}) \in \bar{H}_{DBS}. \quad (4)$$

Because DBS is well-formed for DBS^* , $M \in \Gamma_n$. Thus, no symbol in Σ_n occurs twice in M . Hence, by (4) and construction of $p_{H_{DBS}}[M, R](\text{``}\bar{x}\text{''})$, we have:

$$p_{H_{DBS}}[M, R](\text{``}\bar{x}\text{''}) \in \bar{H}_{DBS}. \quad (5)$$

By Lemma 5.1, we have

$$(\exists (L, S) \in SE) (p_{H_{DBS}}[L, S](\text{``}\bar{x}\text{''}) = p_{H_{DBS}}[M, R](\text{``}\bar{x}\text{''}) \wedge \text{elem}(L) \subseteq \text{elem}(M)). \quad (6)$$

By (6), and Definition 5.2, we have:

$$(M, R) \in WSR. \quad \square \quad (7)$$

Theorem 5.2 has a rather technical flavor, but we can attribute it some practical relevance. We first observe that it depends only on the assumption that GPS has no access to A or DBS^* . Hence it applies to a broader class of schedulers than the one we have been considering. In particular, it also applies to the concurrency control mechanisms of [2, 17]. Thus granted the narrow scope of our underlying model, Theorem 5.2 justifies the almost universal concentration on the notion of serializability as a way to obtain consistency preservation, attested by [2, 3, 7, 9, 16–18, 20, 22].

We close this section by briefly touching on the other synchronization anomalies mentioned in Sect. 2.4, that are not avoided by our correctness criterion (see [4] for a full discussion). Race conditions can be avoided by forcing each log to be equivalent to some serial log containing *all* transactions in question. This condition corresponds to the usual notion of serializability (see, e.g., [7]). An example of a serializable log is given in Ex. 5.1(b). It is easy to see that serializability approximates consistency preservation but, since the log in Ex. 5.1(c) is not serializable, equivalence does not hold.

We guarantee that each transaction T_i will read consistent data by requiring that the log preceding R_i , denoted $L[R_i]$, be weakly serializable. In fact, it suffices to require that $L[R_i]$ produce the same result as a serial log only for the variables in the readset of T_i .

5.2. Conflict Preserving Serializability

In this section we introduce a log predicate, called *conflict preserving serializability* (*CPSR* in [3] or *DSR* in [17]), that induces a strict subset of *WSR*. Hence, *CPSR* only approximates consistency preservation, by the theorems in Sect. 5.1. However, by adapting results from [17], it can be shown that testing for membership in *CPSR* can be done in polynomial time, whereas the same problem is *NP*-complete for *WSR*. Thus, although *WSR* is equivalent to consistency preservation, it is not realistic from a practical point of view and stronger predicates implying *WSR*, such as *CPSR*, have to be used.

The following definition corresponds to Corollary 2 of [17, p. 20].

Definition 5.3. $(L, S) \in \text{CPSR}$ iff $(L, S) \in G$ and there exists a set of distinct real numbers $\bar{p} = \{p_i \in \mathbb{R} / R_i \text{ occurs in } L\}$ such that for all $R_i, W_j, W_k \in \text{elem}(L)$

- (a) if $S(R_i) \cap S(W_j) \neq \emptyset$ and W_j precedes R_i in L , then $p_j < p_i$
- (b) if $S(R_i) \cap S(W_j) \neq \emptyset$, $i \neq j$, and R_i precedes W_j in L , then $p_i < p_j$
- (c) if $S(W_k) \cap S(W_j) \neq \emptyset$ and W_k precedes W_j in L , then $p_k < p_j$.

We say that (L, S) is *conflict preserving serializable* (*CP*-serializable). \square

Example 5.2. (Compare with Ex. 5.1)

- (a) $R_1[x] R_2[y] W_1[x] W_2[x] \in \text{CPSR}$ with $p_1 = 1$ and $p_2 = 2$.
- (b) $R_1[x] R_2[x] W_2[x] W_1[x] \notin \text{CPSR}$ as $p_1 < p_2$, using R_1 and W_2 , and $p_2 < p_1$, using R_2 and W_1 . \square

Definition 5.3 is best understood in terms of the *CP*-serialization of (L, S) induced by \bar{p} , defined as the serial augmented log $(L, S) \in SE$ such that $\text{elem}(L) = \text{elem}(L) - \{R_i \in \Sigma / W_i \notin \text{elem}(L)\}$ and a pair $R_i W_i$ occurs before $R_j W_j$ in L iff $p_i < p_j$. Intuitively, L can be constructed from L by successively switching pairs of adjacent symbols x and y such that $S(x) \cap S(y) = \emptyset$ and then dropping all unmatched read symbols. Note that, since $S(x) \cap S(y) = \emptyset$, the result of the operations represented by x and y is the same whether x is executed before y or vice-versa. Hence, L represents a sequence of operations that performs the same changes on the database as the sequence corresponding to L .

As a consequence of this argument, we can prove that $\text{CPSR} \subset \text{WSR}$, which implies that *CPSR* approximates consistency preservation.

5.3. Extending the Results to Database Systems with Restarts

We now briefly indicate how to extend the results of Sects. 5.1 and 5.2 to systems with restarts. Recall that Γ^0 is the set of all logs such that a read symbol R_i may occur several times before W_i occurs. Also recall that all output logs of a well-formed system with restarts belong to this class.

Given $L \in \Gamma^0$, we say that $L_R \in \Gamma$ is the *reduced log* of L iff L_R is obtained from L by deleting all occurrences of a read symbol except the last. The key observation is that $p_{P^*}[L, S] = p_{P^*}[L_R, S]$, for all interpretations P^* of (L, S) .

We say that $(L, S) \in P$, where P is either *WSR* or *CPSR* iff $(L_R, S) \in P$. Using these extended classes and the equivalence of programs discussed in the previous paragraph, all results of Sects. 5.1 and 5.2 are extended to systems with restarts. However, since systems with restarts may not terminate, we have to assume explicitly that all computations are finite.

6. Examples of General Purpose Schedulers

As promised in previous sections, we now address the question of constructing a general purpose scheduler *GPS* totally correct w.r.t. consistency preservation. Recall that this means that, in any database system using *GPS*, all transactions terminate properly and all computations map the set of consistent database states into itself. By the results in Sect. 5, we can replace consistency preservation either by *WSR* or *CPSR*. Both are in principle easier to use because they impose restrictions directly on the output logs of *GPS*.

All schedulers constructed here will operate based on "plans". A *plan* is a complete log satisfying $P \in \{WSR, CPSR\}$. It corresponds to a scheduling of both the reads and writes of all transactions that have issued a read; the scheduling must ensure P and lead to the proper termination of these transactions. Therefore, if the scheduler follows the current plan and no new transaction issues a read, the final log will coincide with the plan and, hence, will satisfy P . Whenever a new read or write is issued, the plan may be updated according to a fixed strategy. Sometimes the new operation must be delayed to fit into the current plan. In this case, it is enqueued for later rescheduling.

Obviously, the crucial question is how to find plans. Here the type of the database system plays a central role. We show that if the system has no restart facility but writesets are predeclared, then we can exploit the available information to construct a reasonable plan. By this we mean a correct scheduling of operations that achieves a good level of concurrency, intuitively speaking. However, if writesets are not predeclared, we must be so conservative in choosing a plan that the level of concurrency will be very low. The conservatism is required because the plan has to guarantee P for every value the unknown writesets turn out to have. Essentially, this amounts to assuming each transaction writes into the entire database. This problem can be remedied by using restarts. Now, we can adopt the opposite strategy: find a plan under the assumption of an empty writeset. If this assumption leads to an incorrect scheduling, the mistake can be mended by restarting operations.

6.1. Constructing Type 1 General Purpose Schedulers for Systems Without Restarts

Our goal in this subsection is to construct a type 1 general purpose scheduler for systems without restarts totally correct w.r.t. $P \in \{WSR, CPSR\}$. We first generalize the problem somewhat by considering a set of log predicates $PR1$, which includes WSR and $CPSR$. We may view $PR1$ as summarizing the properties of log predicates needed for constructing our schedulers.

Definition 6.1. A log predicate P is in class $PR1$ iff

- (a) $P(A, \emptyset)$
- (b) $(\forall I \subset [1, \infty))(\forall J \subset [1, \infty))(\forall S_I \in \Psi_I)(\forall S_J \in \Psi_J)(\forall L_I \in \Omega_I)$

$$(\exists L_J \in \Omega_J)(I \cap J = \emptyset \Rightarrow (P(L_I, S_I) \Rightarrow P(L_I \| L_J, S_I \cup S_J))). \quad \square$$

Let $TS_J = (J, S_J)$ denote the set of transactions whose readsets and writesets are given by $S_J \in \Psi_J$ (and similarly for I). Then, (a) and (b) (with $I = \emptyset$) imply that we can schedule TS_J so as to ensure P . (b) alone says that if we can schedule TS_I ensuring P , then we can extend the schedule to include TS_J , if $J \cap I = \emptyset$, and still achieve P .

Theorem 6.1. Let $P \in PR1$ and let $GPS_1[P]$ be the type 1 general purpose scheduler for systems without restarts defined in Fig. 2. Then $GPS_1[P]$ is totally correct w.r.t. P . Moreover, if the truth of $\bar{P}(L, S)$ can be tested in polynomial time, then $GPS_1[P]$ is also polynomial, where

$$\bar{P}(L, S) \equiv (\exists L \in \Omega_D)(L \in \text{prefix}(L) \wedge P(L, S)), \quad \text{with } D = \{p \in [1, \infty) / R_p \in \text{elem}(L)\}.$$

We call $GPS_1[P]$ a goal-oriented type 1 general purpose scheduler.

Sketch of Proof. We indicate here how a proof of correctness of $GPS_1[P]$ would be carried out, say, in the formal system of [15]. We state below without proof a set of invariants involving almost exclusively the input and output logs of $GPS_1[P]$. These invariants, as a rule, have to be strengthened to carry out the formal proof (however, they avoid checking interference freedom [15]). Let $DBS = (V, A, n, S, GPS_1[P])$ be a database system and DBS^* be an interpretation for DBS . We have to prove that, for any initial state, DBS halts and the final state satisfies Q , defined as:

$$LI \in \Omega_n \wedge LO \in \Omega_n \wedge P(LO, S) \quad (1)$$

Since DBS has no restarts, it always halts. To prove that Q holds in the final state, we state a series of auxiliary invariants. The first invariant says that LO and $LO1$ are indeed the same:

$$LO = LO1 \quad (2)$$

By the construction of transactions, we know that (3) is an invariant:

$$LI \in \Gamma_n \wedge (\forall p \in [1, n])(W_p = \text{last}(LI) \Rightarrow R_p \in \text{elem}(L1)) \quad (3)$$

The third invariant follows directly from the definition of \bar{P} and $GPS_1[P]$:

$$(\exists L \in \Omega_D)(LO1 \in \text{prefix}(L) \wedge P(L, SO1)) \wedge D = \{p \in [1, \infty) / R_p \in \text{elem}(LO1)\} \quad (4)$$

We now state the definition of $SO1$ and sq and relationships involving LI , $LO1$ and $queue$:

$$SO1 = \{(R_p, S(R_p)) \in S/R_p \in elem(LO1)\} \cup \{(W_p, S(W_p)) \in S/R_p \in elem(LO1)\} \wedge$$

$$sq = \{(R_p, S(R_p)) \in S/R_p \in elem(queue)\} \cup \{(W_p, S(W_p)) \in S/R_p \in elem(queue)\} \quad (5)$$

$$(\forall x \in \Sigma_n)((x = last(LO1) \Rightarrow x \in elem(LI)) \wedge$$

$$(x \in elem(LI) \Rightarrow x \in elem(LO1) \cup elem(queue))) \quad (6)$$

From (4), we have that $LO1 \in \Gamma_n$. Hence, from (4) and (6) we can deduce that DBS is well-formed. That is, we have a sixth invariant:

$$LO1 \in \Gamma_n \wedge (\forall x \in \Sigma_n)(x = last(LO1) \Rightarrow x \in elem(LI)) \quad (7)$$

From (6), we can also deduce that (8) is invariant:

$$(\forall x \in \Sigma_n)(x \in elem(LO1) \Rightarrow x \in elem(LI)). \quad (8)$$

As $LO1$ is modified only by $GPS_1[P]$, it suffices to consider the return state c of an arbitrary call to $GPS_1[P]$. We argue that either $GPS_1[P]$ will not be called again and Q is true in c , or $GPS_1[P]$ will still be called. This argument could be made precise using, say, program counters [8]. However, we leave the discussion of this level, here and in what follows. All formulae below are evaluated in c .

Case 1: suppose that $LO1 \in \Omega_n$.

Then, from (4) and (5) we conclude that $P(LO1, S)$ holds. Now, from (3) and (8) we have that $LI \in \Omega_n$. Therefore, by (2), Q is true. Moreover, as $LI \in \Omega_n$, all transactions have already sent both messages to SCHEDULER. So $GPS_1[P]$ will not be called again.

Case 2: suppose that $LO1 \notin \Omega_n$.

Case 2.1: suppose that there exists $i \in [1, n]$ such that $R_i \notin elem(LI)$.

Then, transaction i will still send a read message to SCHEDULER and $GPS_1[P]$ will be called again.

Case 2.2: suppose that for all $i \in [1, n]$, $R_i \in elem(LI)$.

By (6), for all $i \in [1, n]$, $R_i \in elem(LO1) \cup elem(queue)$.

Case 2.2.1: suppose that $LO1 \in \Omega$.

Then, $LO1 \notin \Omega_n$ implies that, for some $i \in [1, \infty)$, $R_i \in elem(queue)$. But this is not possible, by construction of $GPS_1[P]$ (using (c) of Def. 6.1). More precisely, this assumption contradicts the following invariant:

$$LO1 \in \Omega_n \vee LO1 \notin \Omega \quad (9)$$

which can be proved directly from $GPS_1[P]$ and Definition 6.1, as $P \in PR1$.

Case 2.2.2: suppose that $LO1 \notin \Omega$.

Then, for some $i \in [1, n]$, $R_i \in elem(LO1) \wedge W_i \notin elem(LO1)$, by definition of Ω (see Sect. 3.1).

Case 2.2.2.1: suppose that for some $i \in [1, \infty)$,

$$R_i \in elem(LO1) \wedge W_i \notin elem(LO1) \wedge W_i \notin elem(LI).$$

Then, transaction i will send a write message to SCHEDULER and $GPS_1[P]$ will still be called.

Case 2.2.2.2: suppose that for all $i \in [1, \infty)$,

$$R_i \in \text{elem}(LO1) \wedge W_i \notin \text{elem}(LO1) \Rightarrow W_i \in \text{elem}(LI).$$

Let L_I be the “plan” in state I given by (4). Then, as $LO1_I \notin \Omega$, $L_I = LO1_I \parallel W_k \parallel L'_I$, for some $W_k \in \Sigma_n$ and $L'_I \in \Gamma_n$. By (6) and assumption, $W_k \in \text{elem}(\text{queue})$. But this is not possible, by construction of $GPS_1[P]$. More precisely, this assumption contradicts the following invariant:

$$\begin{aligned} & (\exists L \in \Omega_D)(LO1 \in \text{prefix}(L) \wedge P(L, SO1) \wedge \\ & (\forall L' \in \Gamma_n)(\forall W_p \in \Sigma_n)(L = LO1 \parallel W_p \parallel L' \Rightarrow W_p \notin \text{elem}(\text{queue})) \end{aligned} \quad (10)$$

which can be proved directly from $GPS_1[P]$. \square

$GPS_1[P]$ depends entirely on (a) and (b) of Definition 6.1 and on (c) below, which follows from the fact that all systems in question are type 1:

(c) For any state I , if R_p occurs in $LO1_I(\text{queue}_I)$, then $SO1_I(\text{sq}_I)$ is defined on $R_p, W_p \in \Sigma$.

The output of $GPS_1[P]$ follows a “plan” chosen via \bar{P} . A plan is a complete log L containing all symbols in the domain of $SO1_I$ and corresponds to a scheduling of the operations of all transactions started up to I . Whenever a new symbol is input, say x_I , L may change. If $x_I = R_p$, L may change to accommodate the new transaction T_p that started; if $x_I = W_p$, L may be reorganized to allow W_p to be output. However, the new plan L' is always compatible with L , in the sense that the output log generated up to the current state is still a prefix of L .

Each plan always satisfies P , thus guaranteeing that $GPS_1[P]$ is correct w.r.t. P , if $GPS_1[P]$ is total. Totality is obtained by assuring that a plan always exists and that each symbol in queue will always be rescheduled. This follows by (a) and (b) of Definition 6.1. Furthermore, if queue is not empty and the current plan has been exhausted, it can always be extended to include symbols in queue, by (b).

Fig. 2. The type 1 scheduler $GPS_1[P]$

input	x : a symbol in $\Sigma = \{R_1, W_1, R_2, W_2, \dots\}$ y : if $x = R_p$ then $y = \{(R_p, S(R_p)), (W_p, S(W_p))\}$ else $y = \emptyset$
comment	– x corresponds to the operation sent to the scheduler; $\text{op}.x \in \{‘R’, ‘W’\}$ indicates the operation type and $\text{tr}.x \in [1, \infty)$, the transaction number. – y contains the readset and writeset of T_p , if $x = R_p$, and is \emptyset otherwise. Note that the system is type 1.
output	z : a well-formed (partial) log
comment	z corresponds to the operations scheduled by $GPS_1[P]$.
state variables	$LO1, \text{queue}, q$: strings in Σ^* (initialized to A) $SO1, \text{sq}$: functions from Σ to 2^y (initialized to \emptyset)
comment	– $LO1$ holds the output log created up to the current state. – $SO1$ holds the readset and the writeset of T_p , if R_p occurs in $LO1$. – queue holds the symbols input, but not output up to the current state. – sq holds the readset and the writeset of T_p , if R_p occurs in queue .
$GPS_1[P](x, y; z)$	

```

begin  $z := A$ 
  comment try to output  $x$  according to the strategy dictated by  $\bar{P}$ ;
  if  $\neg \bar{P}(LO1 \parallel x, SO1 \cup y)$ 
    then comment if not possible, enqueue  $x$  for later rescheduling;
      begin queue := queue  $\parallel x$ ;  $sq := sq \cup y$  end
    else begin comment output  $x$ ;
       $LO1 := LO1 \parallel x$ ;  $SO1 := SO1 \cup y$ ;  $z := z \parallel x$ ;
      comment try to reschedule symbols in queue;
       $q := queue$ ; queue :=  $A$ ;
      while  $q \neq A$  do
        begin  $x := \text{car}(q)$ ;  $q := \text{cdr}(q)$ ;  $p := \text{tr}.x$ ;
           $y := \text{if op}.x = 'R'$ 
            then  $\{(R_p, sq(R_p)), (W_p, sq(W_p))\}$  else  $\emptyset$ 
          comment try to output  $x$ 
          if  $\neg \bar{P}(LO1 \parallel x, SO1 \cup y)$ 
            then queue := queue  $\parallel x$ ;
            else begin  $LO1 := LO1 \parallel x$ ;  $SO1 := SO1 \cup y$ ;
               $z := z \parallel x$ ;  $sq := sq - y$ 
            end
          end
        end
      end
    return  $z$ 
  end
end

```

We close this section by briefly investigating the time complexity of $GPS_1[WSR]$ and $GPS_1[CPSR]$. The crucial step is obviously testing \bar{P} . Since testing for membership in WSR is NP -complete, then clearly testing \bar{WSR} is also NP -complete. Therefore $GPS_1[WSR]$ will almost certainly not be practical. On the other hand, it is shown in [4] that there is an efficient algorithm testing \bar{CPSR} , thus making $GPS_1[CPSR]$ quite practical and justifying our introduction of $CPSR$.

6.2. Constructing Type 0 General Purpose Schedulers for Systems Without Restarts

We now discuss the feasibility of constructing type 0 general purpose schedulers for systems without restarts. Consider first extending Theorem 6.1 to type 0 systems. As the writesets are not predeclared in type 0 systems we cannot find a plan L such that L is complete and SO_l is defined on all symbols of L .

One way to circumvent this problem, without changing the definition of class $PR1$, would be to find a plan L such that $P(L, SO_l \cup S')$ holds for all possible values for the unknown writesets. That is, construct a type 0 general purpose scheduler $GPS_0[P]$ similar to $GPS_1[P]$, but redefine \bar{P} as follows:

$$\bar{P}(L, S) \equiv (\exists L' \in \Omega_D)(\forall S' \in \Phi) \quad (1)$$

$$(\text{DOMAIN}(S') = \Sigma_D - \text{DOMAIN}(S) \Rightarrow L \in \text{prefix}(L) \wedge P(L, S \cup S'))$$

where

$$D = \{p \in [1, \infty) / R_p \in \text{elem}(L)\}$$

A result similar to Theorem 6.1 could be stated for $GPS_0[P]$.

However, for exactly the same reasons as in Sect. 6.1, we can dismiss $GPS_0[WSR]$ as not practical. So let us concentrate on $GPS_0[CPSR]$. Testing the truth of $\overline{CPSR}(L, S)$ can again be done in polynomial time, because the following equivalence holds:

$$\overline{CPSR}(L, S) \equiv (\exists L' \in \Omega_D)(L \in \text{prefix}(L') \wedge (L, S \cup S_w) \in CPSR) \quad (2)$$

where

$$S_w = \{(W_p, V) \in \Sigma \times \{V\} / W_p \in \Omega_D - \text{DOMAIN}(S)\}$$

and D is given as in (1).

(2) says that we can replace the quantification over S' in (1) by the assumption that each unknown writeset is equal to V , the set of database variables. This assumption generates the maximum number of possible "conflicts", if one understands each condition in the definition of $CPSR$ as defining a conflict ((a), (b) and (c) in Def. 5.3). Putting it differently, $GPS_0[CPSR]$ must be so conservative to the point of impairing concurrency.

We go even further and prove that *any* type 0 general purpose scheduler for systems without restarts, say GPS_0 , that is totally correct w.r.t. $CPSR$ cannot achieve a reasonable level of concurrency. Consider first the following example.

Example 6.1. (a) Consider a type 0 system without restarts and the following sequence of partial logs (using the notation of Example 5.1):

Partial Input Log	Partial Output Log	Queue for Later Scheduling
$R_1[x]$	$R_1[x]$	
$R_1[x] R_2[y]$	$R_1[x] R_2[y]$	
$R_1[x] R_2[y] W_2[x, y]$	$R_1[x] R_2[y]$	$W_2[x, y]$
$R_1[x] R_2[y] W_2[x, y] W_1[x, y]$	$R_1[x] R_2[y]$	$W_2[x, y] W_1[x, y]$.

Suppose that the system has just two transactions. Then, either a non CP -serializable or an incomplete final output log will result. In fact, no correct scheduler (w.r.t. CP -serializability) could have output R_2 after R_1 without knowing W_1 , for otherwise the above situation might happen. \square

Briefly, Ex. 6.1 tells us the following fact about a computation of GPS_0 :

(1) Suppose that R_i and R_j have been output, but W_i and W_j have not been input. Then, if $S(R_i) \neq \emptyset$ and $S(R_j) \neq \emptyset$, we can take $S(W_i) = S(W_j) = S(R_i) \cup S(R_j)$ so that no complete output log will be CP -serializable.

(1) follows directly from the definition of $CPSR$. Likewise, we have that:

(2) Suppose that R_i, R_j and W_j have been output, but W_i has not been input. Then, if $S(R_i) \cap S(W_j) \neq \emptyset$ R_i precedes W_j in the output log, we can take $S(W_i) = S(W_j)$ so that no complete output log will be CP -serializable.

By (1) and (2), GPS_0 must then satisfy condition Q below:

Q . if $x \in \Sigma$ is output, then $S(x) \cap S(R_i) = \emptyset$, for all transactions T_i such that R_i has been output, but W_i has not been input.

A more precise statement of the above restriction is that Q has to be an invariant of GPS_0 .

Theorem 6.2. Let GPS_0 be a type 0 general purpose scheduler for systems without restarts. Suppose GPS_0 is total. If GPS_0 is correct w.r.t. $CPSR$, then Q is an invariant of any database system $DBS \in db_{S_0}[GPS]$ with any interpretation DBS^* , where

$$Q. (\forall j \in [1, \infty)) (\forall i \in [1, \infty)) (\{i, j\} \subseteq ACTIVE \Rightarrow SI(R_i) \cap SI(R_j) = \emptyset) \wedge \\ (\forall j \in [1, \infty)) (\forall i \in [1, \infty)) (W_j \in elem(LO) \wedge i \in ACTIVE \Rightarrow SI(R_i) \cap SI(W_j) = \emptyset)$$

with

$$ACTIVE = \{i \in [1, \infty) / R_i \in elem(LO) \wedge W_i \notin elem(LI)\}.$$

Sketch of Proof. The proof uses (1) and (2) of the preceding discussion to derive a contradiction. \square

Theorem 6.2 is much more powerful than our preceding results because it says that no reasonable type 0 scheduler for systems without restarts totally correct w.r.t. $CPSR$ can be built, assuming our model or any other model that subsumes ours. This interpretation obviously depends on the metric for concurrency one has in mind or, rather, on the notion of scheduler performance.

6.3. Constructing Type 0 General Purpose Schedulers for Systems with Restarts

This section briefly sketches a type 0 general purpose scheduler for systems with restarts correct w.r.t. $P \in \{WSR, CPSR\}$, called $GPSR_0[P]$, using techniques quite similar to those of Section 6.1. To motivate the discussion, let us consider an example showing how restarts can be used to circumvent the problems raised in Section 6.2 about $GPS_0[CPSR]$.

Example 6.2. Consider the same initial situation as in Ex. 6.1, but suppose that the system has a restart facility. Then, the following would be a feasible scenario:

Partial Input Logs

$R_1[x]$
 $R_1[x] R_2[x]$
 $R_1[x] R_2[x] W_2[x, y]$
 $R_1[x] R_2[x] W_2[x, y] W_1[x, y]$
 $R_1[x] R_2[x] W_2[x, y] W_1[x, y] W_1[x, y]$

Partial Output Logs

$R_1[x]$
 $R_1[x] R_2[x]$
 $R_1[x] R_2[x] W_2[x, y]$
 $R_1[x] R_2[x] W_2[x, y] R_1[x]$
 $R_1[x] R_2[x] W_2[x, y] R_1[x] W_1[x, y]$

W_2 is output on the third step under the assumption that $S(W_1) = \emptyset$, instead of $S(W_1) = V$ as in Sect. 6.2. When $S(W_1)$ becomes known in the fourth step, the scheduler discovers that the current output log is not a prefix of any log in $CPSR$. R_1 is then restarted (repeated in the output log) and W_1 ignored. W_1 is output when the scheduler receives it again. \square

As in Sect. 6.1, we generalize the problem by considering a class of log predicate $PR2$, which includes WSR and $CPSR$.

Definition 6.2. A log predicate P is in class $PR2$ iff

- (a) and (b) – as in Definition 6.1;

$$(c) (\forall I \subset [1, \infty)) (\forall S \in \Psi_I) (\forall L \in \Omega_I) (\forall W_p \in \text{elem}(L)) \\ (S(W_p) = \emptyset \Rightarrow P(L, S) \Rightarrow P(\text{delete}(\{R_p, W_p\}, L), S - \{(R_p, S(R_p)), (W_p, S(W_p))\}))$$

where $L' = \text{delete}(X, L)$ iff L' is obtained from L by deleting every symbol in $X \cap \text{elem}(L)$ from L . \square

In words, to build $GPSR_0[P]$, we need the same assumptions about P as in Sect. 6.1, plus a third one. (c) says that if L is a schedule for the set of transactions TS_I , then $L' = \text{delete}(\{R_p, W_p\}, L)$ is a schedule for TS_J , where TS_J contains all transactions in TS_I , except T_p .

We now briefly describe how $GPSR_0[P]$ would work (a more precise description can be found in [4]). Recall that $GPS_0[P]$ constructed plans based on the assumption that each unknown writeset was equal to V , the set of database variables. $GPSR_0[P]$ replaces this conservative assumption by an optimistic one: assume the unknown writesets to be empty. Therefore, in $GPSR_0[P]$, \bar{P} is redefined as:

$$\bar{P}(L, S) \equiv (\exists L' \in \Omega_D) (L \in \text{prefix}(L') \wedge P(L, S \cup S_w))$$

where

$$S_w = \{(W_p, \emptyset) \in \Sigma \times \{\emptyset\} / W_p \in \Sigma_D - \text{DOMAIN}(S)\}$$

and

$$D = \{p \in [1, \infty) / R_p \in \text{elem}(L)\}.$$

When a write symbol W_p is input for the first time, $S(W_p)$ becomes available. The current tentative plan L is then revised on the basis of the actual writeset $S(W_p)$. If no plan L' exists such that the current output log is a prefix of L' , L is considered erroneous. R_p is then restarted, that is, deleted from the current output log and added to *queue*. $S(W_p)$ is saved and used later to plan the rescheduling of R_p so that R_p is not restarted again. (Therefore, it is crucial to assume that $S(W_p)$ will not change when T_p is reexecuted. Alternatively, in a more general model, there must be a way of guaranteeing that each operation will not be indefinitely restarted [18].)

Total correctness is argued as in Sect. 6.1. In addition, we have to use (c) of Definition 6.2 to guarantee that when R_p is deleted from the current output log L , a new plan L' exists. L' is created by deleting R_p and W_p from the old plan L . Note that, since L was a prefix of L' , L with R_p deleted is a prefix of L' .

We close this section by observing that the time bounds of $GPSR_0[P]$ and $GPS_1[P]$, $P \in \{WSR, CPSR\}$, are identical.

7. Conclusions

We explored some issues raised by the construction of correct database systems, using very simplistic models. We concentrated exclusively on consistency preservation, since this criterion is universally adopted and creates interesting problems. When passing from database systems to general purpose schedulers, we were able to justify the shift from consistency preservation to serializability usually taken. In fact, one contribution of this work was to link system correctness criteria

with scheduler properties. Finally, exploring our different models, we indicated a basic trade-off in the construction of efficient schedulers: either the scheduler must be able to plan ahead using advanced information about future operations or some mechanism must be available to reorder past operations.

Acknowledgments. This work was supported by the Conselho Nacional de Pesquisas, Brazil, grant No. 1112.1248/76 to M.A.C. and in part by the National Science Foundation, grant No. MCS-77-05314 to P.A.B.

References

1. Aho AV, Hopcroft JE, Ullman JD (1975) The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass
2. Bernstein PA, Shipman DW, Rothnie JB (1980) Concurrency control in a system for distributed databases (SDD-1), ACM TODS 5: 18-51
3. Bernstein PA (1979) A formal model of concurrency control mechanisms for database systems, IEEE Transactions on Software Engineering p 203
4. Casanova MA, Bernstein PA (1979) General Purpose Schedulers for database systems, Technical Report TR-08-79, Center for Research in Computing Technology, Harvard University
5. Casanova MA, Bernstein PA (1980) On the construction of database schedulers based on conflict-preserving serializability. (In press)
6. Cook SA (1975) Axiomatic and interpretative semantics for an ALGOL fragment, TR-79, Univ of Toronto
7. Eswaran KP, Gray JN, Lorie R, Traiger I (1976) The notion of consistency and predicate locks in a data base system, CACM 19:624-633
8. Flon L, Suzuki N (1977) Nondeterminism and the correctness of parallel programs, IFIP Working Conference on the Formal Description of Programming Concepts
9. Gardarin G, Lebeaux P (1977) Scheduling algorithms for avoiding inconsistency in large databases, Proc of the Int Conf on Very Large Data Bases, 1977, p. 501
10. Howard JH (1976) Proving monitors, CACM 19: 273-279
11. Kung HT, Papadimitriou CH (1979) An optimality theory of concurrency control for databases, ACM-SIGMOD 1979 International Conference on Management of Data, ACM, NY, p 116-126
12. Lamport L (1976) Towards a theory of correctness for multi-user data base systems. Massachusetts Computer Associates Tech Rep CA-7610-0712
13. Lorie RA (1977) Physical integrity in a large segmented database, ACM TODS 2:91-104
14. Manna Z (1974) Mathematical theory of computation, McGraw-Hill, New York (Chap 4 pp 260)
15. Owicki SS (1975) Axiomatic proof techniques for parallel programs, Dept of Comp Science, Cornell Univ, Tech Rep 75-251
16. Papadimitriou CH, Bernstein PA, Rothnie JB (1977) Some computational problems related to database concurrency control, Proc of Conf on Theoretical Computer Science, Waterloo, Aug 1977, pp 275-282
17. Papadimitriou CH (1979) The serializability of concurrent updates, JACM 26: 631-653
18. Rosenkrantz DJ, Lewis PM, Stearns RE (1978) System level concurrency for distributed database systems, ACM TODS 3: 178-198
19. Schlageter G (1978) Process synchronization in database systems, ACM TODS 3: 248-271
20. Stearns RE, Lewis PM, Rosenkrantz DJ (1976) Concurrency control for database systems, Proc of the 17th IEEE Symp on Found of Comp Sci, pp 19-32
21. Stonebraker M, Neuhold E (1977) A distributed database version of INGRES, Proc 2nd Berkeley Workshop on Distributed Databases and Computer Networks, Berkeley, California
22. Thomas RH (1979) A majority consensus approach to concurrency control for multiple copy databases, ACM TODS 4: 180-209

Received July 2, 1979; Revised March 3, 1980