Semantic Serializability: A Correctness Criterion for Processing Transactions in Advanced Database Applications

A. Brayner T. Härder N. Ritter

University of Kaiserslautern
Dept. of Computer Science - AG DBIS
P.O. Box 3049
D-67653 Kaiserslautern - Germany
E-Mail: {brayner,haerder,ritter}@informatik.uni-kl.de

Abstract

Serializability requires that the execution of each transaction must give the illusion to be an atomic action, i.e., the execution of a set of transactions must appear to be a serial one. This requirement, however, is too strong and unnecessarily restricts concurrency among transactions, when semantic information is available for the transaction processing mechanism. In this paper, a new correctness criterion for concurrent execution of database transactions, denoted *semantic serializability*, is proposed. *Semantic serializability* is based on the use of semantic information about database objects (and not about transactions). The main idea of our proposal is to provide different atomicity views for each transaction and, for this reason, to allow interleavings among transactions which are nonserializable, but which preserve database consistency. We develop two concurrency control protocols, which are based on *semantic serializability*. One protocol is based on a locking-mechanism and the other one uses a non-locking approach. Our proposal is suitable to a wide variety of advanced database applications, such as CAx, MDBS, GIS and WFMS.

Keywords: Long-living transaction; Semantic knowledge; Concurrency control; Schedule; Consistency; Serializability

1 Introduction

Database states represent snapshots of the real world. They only reflect static aspects of the world. Changes in the real world are represented in databases by state transitions. State transitions are realized by application programs which contain database operations (read and write operations on database objects). In practice, application programs run concurrently. Evidently, some interleaved executions of steps of different programs may lead to inconsistent changes in the database. In order to avoid inconsistencies, the DBMS must monitor and control concurrent executions of multiple application programs. However, from the concurrency control perspective, not all operations of a program are relevant, only database operations are to be considered. To represent the database actions of a program, we use the concept of transaction. *A transaction is an abstraction that represents a sequence of database operations resulting from the execution of an application program.*

In the conventional model for database concurrency control, a transaction represents three distinct units. A transaction represents a *logical unit*, in the sense that the execution of a transaction has its own local context, in which the results of earlier operations are maintained to be used for later actions of that transaction. Regarding its local context a transaction is executed in isolation from others. The execution of a transaction must give the illusion to be carried out as an atomic action, without interference or interleaving from steps of other transactions. For this reason, we consider a transaction being an *atomic unit*. Finally, a transaction represents a *recovery unit*, that is, if its execution is interrupted for some reason and cannot be completed, the effect of its initial actions must be wiped out. However, these units do not necessarily have to be represented as a transaction. To illustrate this fact, consider the SAGAS model [12]. Here, a transaction, denoted *saga*, represents a logical unit, whereas its subtransactions represent atomic units. The SAGAS model supports two different units for recovery: a saga or a subtransaction.

In the classical model for transaction management, the requirement that transactions are atomic is enforced by *"syntactic" serializability* [9, 22]. *Strictness* [4, 15] enforces that a transaction represents a recovery unit.

This model has been quite appropriate for conventional database applications in which transactions are typically of short duration and are executed within a few seconds. However, in recent years, database concepts have been applied to areas such as computer-aided design and software engineering (CAD and CASE), geographic information systems (GIS) and workflow management systems (WFMS). These new database applications consist of long-living and (possibly) interactive transactions.

In existing databases, the requirements of atomicity and strictness were implemented by locking mechanisms. In such mechanisms, a lock on a database object must be held by a transaction until it commits (or aborts). Clearly, waiting for such a lock may cause unacceptable delays in applications with long-living transactions. Additionally, considering a long transaction as a recovery unit may result in an unacceptable loss of work in the case of aborts or system crashes. Therefore, the classical transaction model is inadequate for such non-conventional database applications which basically consist of long-living activities.

It is obvious that applications with long-living transactions need a new concept for transaction management. This new concept should introduce more *intertransaction parallelism* and, consequently, more concurrency among the long-living activities. This can only be achieved, if the classical notions of atomicity and recoverability are adjusted to the requirements of the new database applications. More precisely, atomic units and recovery units should be refined, i.e.: *atomic unit* "$\leq$" *recovery unit* "$<$" *transaction*.

A recovery unit within a transaction is embodied by a sequence of operations, which guarantees that a consistent database state is obtained by the operations within these units. Existing approaches supporting recovery units smaller than the atomic unit require that the user correctly specifies recovery unit boundaries. As we will see, our approach considers recovery units consisting of at least one atomic unit and, for that reason, allows the automatic determination of recovery unit boundaries by the transaction mechanism. For that reason, we claim that the optimal granularity for those units is the following: *atomic unit* "$\leq$" *recovery unit*.

In this paper, we propose a new model for database concurrency control relaxing absolute atomicity of transactions. A transaction may consist of more than one atomic units. The determination of the atomic units belonging to a transaction is done "on-the-fly" by schedulers. Thus, users do not need to define atomic units. Our proposal introduces a new notion of correctness for schedules, denoted *semantic serializability*, as well as a new notion of equivalence of schedules, which are based on the use of semantic knowledge about database objects. Our proposal also supports an automatic definition of recovery units. Moreover, two concurrency control protocols are presented, which implement *semantic serializability* as our criterion to produce correct schedules. One protocol is based on a locking-mechanism and the other one uses a non-locking approach.

This paper is organized as follows. In the next section, we review some proposals which extend the conventional concepts of atomicity and recoverability. In section 3, we motivate the applicability and feasibility of our proposal by describing a concrete example of a VLSI CAD application. In Section 4, we present a formal model for transaction processing which is based on the relaxation of transaction atomicity. A new correctness criterion and an efficient method to identify correct schedules are presented. In Section 5, we describe two concurrency control protocols for our transaction processing model whereas Section 6 discusses the applicability of our proposal. Finally, Section 7 concludes the paper.

2 Related Work

Many researchers started to extend the conventional concepts of atomicity and recoverability. One approach used to address this problem is to place more structure on transactions. The nested transaction model [21] has been following this approach. Here a transaction may consist of a hierarchy of subtransactions, where each subtransaction represents an atomic unit (w.r.t. sibling-transactions) and a recovery unit as well. An extension to this model is the open nested transaction model [27]. In this model, each subtransaction may represent an atomic unit w.r.t. root transactions. However, these approaches still use serializability at every level of the transaction hierarchy.

Shasha proposes in [24] to relax transaction atomicity by chopping transactions into smaller pieces. In order to decompose a transaction correctly, the chopping procedure proposed in [24] assumes that all operations of the transaction are previously known by the procedure. Hence, the chopping process must be realized before transactions are executed. That means, transaction chopping is a static process. In many advanced database applications, transactions are interactive, for example, transactions in CAD and CASE applications. Their operations are specified *ad hoc* by users. Consequently, the chopping process can not be applied to such transactions. Furthermore, the chopping mechanism presumes that every database access performed by a transaction T_i is executed exactly in one piece. That is, a write operation w_i and the read operations whose values determine the result of w_i must be executed within the same piece. However, in [24] it is not detailed how corresponding dependencies between database operations can be identified and, therefore, it is not clear how the enhanced requirement can be enforced.

Some researchers have followed another approach to relax absolute atomicity of transactions. They have considered the use of semantic information about transactions or their operations. For example, several papers [1, 10, 11, 20, 26] have proposed new notions of atomicity and new criteria for correct interleaved execution of transactions. In these proposals, the user must identify and specify the atomic units of a transaction relative to each other transaction of an application (as in [1, 10, 20, 26]) or relative to a set of transactions (as in [11]). This means that transactions may have varying granules of atomic units to be specified relative to each other.

In fact, when the user defines the atomic units of a transaction t_i w.r.t. a transaction t_j, he implicitly specifies in which way steps of t_i may be *correctly* interleaved by steps of t_j. The scheduler is completely dependent on this kind of semantic information (given by the user) to enforce a correct interleaved execution of transactions (i.e., correct schedules). This requirement represents a serious drawback of such proposals. For instance, suppose an application with 100 transactions, where each transaction may have three interleaving points relative to each other transaction. The user must specify 29700 interleaving points for this application. Besides being a cumbersome activity for the user, there is a high probability for erroneous specifications. The incorrect specification of just one interleaving point from the set of 29700 points is a necessary and sufficient condition for database inconsistency.

As already said, semantic serializability relaxes absolute atomicity of transactions. It exploits the dynamic aspect of atomic units presented in [1, 10, 11, 20, 26]. However, our model extends this notion in two main aspects. First, our proposal supports automatic identification of correct interleaving points at run time. Note that transactions are not divided into smaller pieces. This is a very important property, because schedulers realizing our model dynamically identify interleavings which may represent non-serializable schedules but, nevertheless, preserve consistency. Thus, semantic serializability can also be applied to interactive transactions. Second, our approach exploits the use of semantic information about database objects (and not about transactions or their operations).

3 Motivating Example

We motivate the applicability and feasibility of our proposal by describing an example of a simple VLSI CAD application. We use this scenario as a running example throughout the paper in order to illustrate definitions and statements.

First, we briefly want to characterize a typical VLSI CAD design environment. The specification of a circuit consists of two components [3]. One component is the *interface* which represents the functional and input/output (I/O ports) specifications. The second component is the *implementation* which specifies the configuration of circuit elements and their interconnections. Each element, in turn, is associated with an interface and implementation. Thus, the circuit has a hierarchical structure.

As a concrete example we consider the following design scenario. A circuit manufacturer performs an ambitious project to innovate the architecture of two different circuits Circuit C1 and Circuit C2. Two designers, John and Alice, are responsible for the development of the new concepts. Figure 1 illustrates the organization of the project. The continuous lines represent the different components of the circuits, which are named A, B, C and so forth. The dotted lines represent the assignment of tasks to the designers. As observed in [2], in complex design projects, several tasks (subcontracts in [2]) can be assigned to a single designer. For example, the specification of the interface and the implementation of the Circuit C1 (components A and B) is assigned to John, whereas Alice is responsible for the design of some components of Circuit C1. Thus, the specification of components F, G and H was allocated to Alice. It is important to notice that Alice may read the specifications of component E (the interface of F) in order to design component F, although she has not been allocated to design E. That means, Alice has not the grant to modify E. Hence, she may read the component E, but she may not update it.

$\mathcal{DB}$ is the CAD database used in the project described above. In order to facilitate the understanding of the examples used throughout this paper, we make the following simplifications: First, we consider elements of circuits as database objects. Thus: $\mathcal{DB}$={A,B,C,D,E,F,G,H,P,U,V,X,Y,Z}. Second, the *checkout* and *checkin* operations [18] on design objects are represented by simple *read* and *write* operations.

We have observed that CAD applications often "see" the database as a collection of disjoint groups of design objects. Furthermore, we have identified the following property of such applications: *Objects of different groups are completely unrelated with respect to update operations, that is, an object of a group will never be updated on the basis of values of objects belonging to other groups.* For instance, the CAD application of our project example (Fig. 1) "sees" the database $\mathcal{DB}$ as a collection of two disjoint groups of objects: a group consisting of design objects for Circuit C1 and another group for Circuit C2. Obviously, the final result of an update operation on a component of Circuit C2 will by no means be determined by a value read from a component of Circuit C1 and vice-versa.

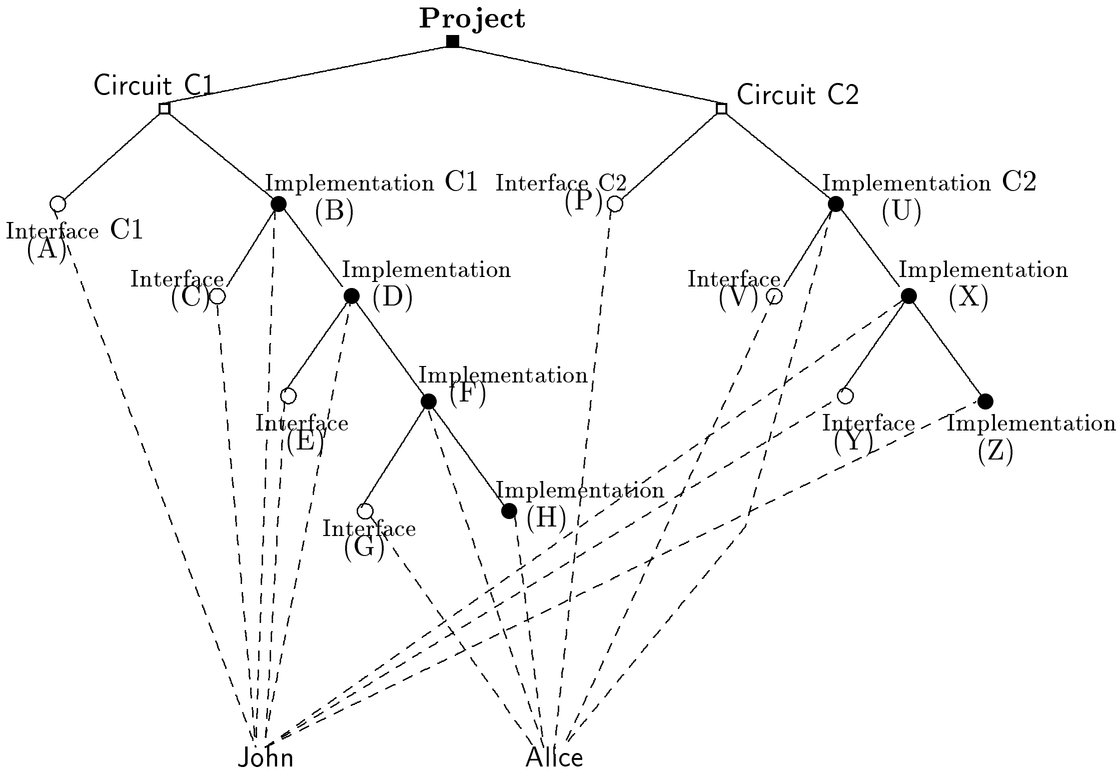

Figure 1: Organization of the project example.

We have investigated the use of such semantic knowledge in transaction processing in order to introduce a higher degree of parallelism and, consequently, more cooperation among long-living transactions. As result of our examinations, we propose a new transaction model which is based on the use of this semantic knowledge and, thereby, offers the mentioned benefits.

It is worthwhile to notice that for our proposal the database does not need to be physically partitioned in different groups of design objects. Partitioning is logical and it is sufficient that only the scheduler has corresponding knowledge.

4 The Model

The gist of our proposal is to use the fact that many non-conventional database applications "see" the database as a collection of logical disjoint groups of objects and that there are no update dependencies between objects of different groups. Using this semantic knowledge, we refine the unit atomicity and, thereby, increase parallelism among transactions. In this section, we formalize the use of this kind of semantic knowledge by introducing a formal model for transactions as well as a corresponding correctness criterion.

4.1 Semantic Units

As shown in our project example, the database may be viewed by many CAD applications as a collection of "semantic islands", each of which representing a collection of objects (entities of the real world). As already mentioned, there are no update dependencies among these "semantic islands". We call those semantic islands *semantic units*.

Before formalizing the notion of semantic unit, we need to specify an additional concept denoted *depends on*. It is derived from the notion of *dependence relation* between the (final) result of updating an object x and the value of object y. We say that x *depends on* y, if the result of any update operation on x is a function of (i.e. is depending on) a value of y read in any program of an application that accesses $\mathcal{DB}$, where $x, y \in \mathcal{DB}$. *depends-on-set(x)* represents the set of all objects *x is depending on.*

Definition 1. Let $\mathcal{DB}$ be a database. SU_i, $0 < i \leq n$, are semantic units of $\mathcal{DB}$, iff
(i) $\mathcal{DB}= \bigcup_{i=1}^{n} SU_i$,
(ii) $\forall 1 \leq i, j \leq n, i \neq j : SU_i \bigcap SU_j = \emptyset$, and
(iii) $(\forall x \in SU_i, y \in SU_j, i \neq j) \Rightarrow (x \notin \text{depends-on-set}(y)) \wedge (y \notin \text{depends-on-set}(x))$ $\diamond$

Intuitively, condition (iii) says that updates on objects of a semantic unit only depend on values of objects of the same semantic unit.

Example 1. Consider the database $\mathcal{DB}$={A,B,C,D,E,F,G,H,P,U,V,X,Y,Z} of our project example (Sec. 3). As already said, the CAD application sees $\mathcal{DB}$ as a set of two disjoint groups of objects: a group of design objects for Circuit C1 and another group for Circuit C2. Moreover, one can easily observe that the final result of an update operation on an object of Circuit C2 will never be determined by a value of an object of Circuit C1 (i.e., no object of Circuit C2 *depends* on objects of Circuit C1) and vice-versa. Thus, two semantic units may be specified for our project example:
SU_{C1} ={A,B,C,D,E,F,G,H} representing the set of design objects of Circuit C1, and
SU_{C2} ={P,U,V,X,Y,Z} representing the set of design objects of Circuit C2. $\diamond$

Obviously, if we want to exploit the notion of semantic units in transaction management, the scheduler must receive information (from users) about the specification of semantic units. One possibility is to describe the semantic units as metadata in the DB-schema and to give the scheduler the ability to obtain this information. An alternative is a tailored specification of semantic units for the scheduler. In both cases our approach consists of having a scheduler which may support the classical transaction model, if no information about semantic units is disposable.

4.2 Transactions

A transaction is modeled as a finite sequence of *read* and *write* operations on database objects. We use the notation $r_i(x)$ ($w_i(x)$) to represent a read (write) operation executed by a transaction T_i on object x. $OP(T_i)$ represents the set of operations executed by T_i. $OP_{SU_a}(T_i)$ represents the set of operations of T_i which are executed on objects of the semantic unit SU_a. With $<_{T_i}$, we denote the precedence relation (execution order) between two operations in a transaction T_i. For instance, $r_i(x) <_{T_i} r_i(y)$ denotes that $r_i(x)$ precedes $r_i(y)$ or $r_i(x)$ is executed before $r_i(y)$. It is assumed that the execution of a transaction preserves database consistency, if the transaction runs in isolation of other transactions.

In the classical transaction model, a write operation in a transaction T is potentially a function of *all previously read values of database objects* in T [23]. However, if the notion of semantic units is used, the condition *(iii)* of Definition 1 supports the following property: a write operation in a transaction T is only a function of *previous read operations on objects which belong to the same semantic unit of the written object*. Hence, the notion of semantic units embeds information which can be exploited by the transaction processing mechanism. Next, we describe the semantic knowledge which is associated with semantic units and may be utilized in transaction management to increase concurrency.

Definition 2. Let $SU_a \subseteq \mathcal{DB}$ be a *semantic unit* of a database $\mathcal{DB}$ and x an object of $\mathcal{DB}$, where $x \in SU_a$. We define the set of *semantically relevant previous reads* of a write operation $w_i(x)$ in a transaction T_i as follows:

$$s\_pre\_read(w_i(x)) := \{r_i(y) : (r_i(y) <_{T_i} w_i(x)) \wedge (y \in SU_a)\} \qquad \diamond$$

Next, we define the semantics of write operations in a transaction. We do this using the notion of *Herbrand semantics* [15, 22].

Definition 3. Let T be a transaction. The *Herbrand semantics* of a $w(x) \in OP(T)$, denoted $\mathcal{H}_T(w(x))$, is defined as follows:

$$\mathcal{H}_T(w(x)) := f_{w(x)}(r(y_1), r(y_2), \ldots, r(y_m)), \quad \text{where } r(y_k) \in s\_pre\_read(w(x)), 0 < k \leq m \qquad \diamond$$

This definition incorporates the property supported by the third condition of Definition 1 in the semantics of transactions. Furthermore, it shows that our model guarantees a more precise identification of the read operations on which a write operation may depend.

Definition 4. Let T be a transaction on database $\mathcal{DB}= \bigcup_{i=1}^{n} SU_i$, where $SU_i, 0 < i \leq n$ are semantic units of $\mathcal{DB}$. For each $OP_{SU_i}(T) \neq \emptyset$, we define an *atomic unit* for T, denoted $\Pi_{[OP_{SU_i}(T)]}(T)$. Each atomic unit $\Pi_{[OP_{SU_i}(T)]}(T)$ consists of the subsequence of operations in T executed on objects of SU_i, where:

$$\forall p, q \in OP_{SU_i}(T),\ p <_{\Pi_{[OP_{SU_i}(T)]}(T)} q \Leftrightarrow p <_T q \qquad \diamond$$

Consider the following transaction which represents design actions executed in our project example (Section 3): $T_{Alice} = r_{Alice}(E) r_{Alice}(P) w_{Alice}(F) w_{Alice}(U) w_{Alice}(V)$
Observe that the transaction T_{Alice} consists of two atomic units. One atomic unit represents the subsequence of operations on objects of the semantic unit SU_{C1} (Example 1) and is represented as follows:

$$\Pi_{[OP_{SU_{C1}}(T_{Alice})]}(T_{Alice}) = r_{Alice}(E) w_{Alice}(F)$$

The other atomic unit for T_{Alice} consists of operations on objects of semantic unit SU_{C2}:

$$\Pi_{[OP_{SU_{C2}}(T_{Alice})]}(T_{Alice}) = r_{Alice}(P) w_{Alice}(U) w_{Alice}(V)$$

Obviously, in our approach a transaction may consist of several atomic units. However, the crucial difference between our approach and the proposals presented in [20, 11, 10, 1, 26] is that the atomic units of a transaction may be identified by the scheduler automatically and dynamically (at run time). In our approach, the user does not have to specify atomic units for each (pair of) transaction(s) as necessary in the cited proposals. Moreover, transactions are not divided into smaller pieces as proposed in [24]. Thus, the scheduler can automatically (and dynamically) identify correct interleaving points. In section 4.3, we discuss the corresponding algorithm.

Next, we want to identify a new transaction property helping to exploit the benefits of our model. We call this property *semantic commutativity*. The basic idea behind this concept is the following. As already said, no update operation on objects of a particular semantic unit depends on values of objects of other semantic units (Definition 1.*(iii)*). Hence, two operations on objects of different semantic units in a transaction T may be commuted (inside T) without changing the effect of T on the database. For example, we may commute the operations $r_{Alice}(P)$ and $w_{Alice}(F)$ in the transaction T_{Alice} leading to the following operation sequence:

$$T'_{Alice} = r_{Alice}(E) w_{Alice}(F) r_{Alice}(P) w_{Alice}(U) w_{Alice}(V)$$

One can easily observe that, when T_{Alice} and T'_{Alice} are executed on the same initial state, they always produce the same state, although T_{Alice} and T'_{Alice} represent different transactions (they have the same set of operations but different sequences). We say that T_{Alice} and T'_{Alice} are *semantically identical*.

Next, we describe what we call well-formed (well-behaved) transactions in our model. This notion is used in the following section to introduce correct schedules.

Definition 5. A transaction T is characterized as *module-structured*, if operations of T are partitioned in subsequences, denoted as *modules*, such that each module represents an atomic unit of T. $\diamond$

Intuitively, a *module-structured* transaction represents a sequence of modules, where each module encompasses operations on objects of only one semantic unit. Moreover, the operations on objects of a semantic unit appear in only one module. For example, the following transaction may be characterized as being module-structured:

$$T'_{Alice} = \underbrace{r_{Alice}(E)w_{Alice}(F)}_{module}\overbrace{r_{Alice}(P)w_{Alice}(U)w_{Alice}(V)}^{module}$$

Observe that the modules of a *module-structured* transaction are in fact atomic units. Here it is important to note that the notion of *module-structured* transactions reflects a transaction property. By this notion, we want to represent the fact that some transactions present a serial execution of atomic units without interleavings of operations belonging to different atomic units. It does not mean that our model requires that transactions should be partitioned into smaller pieces as proposed in [24].

4.3 Correct executions of concurrent transactions

Transactions are executed concurrently. In transaction processing, the parallelism in the execution of several transactions is realized by interleaving transactions. The execution of several interleaved transactions is modeled by *schedules*. Formally, a *schedule* over $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ represents an interleaved sequence of operations of transactions in $\mathcal{T}$ which is an element of the *shuffle* product $T_1 * T_2 * \cdots * T_n$. $OP(S)$ denotes the set of operations executed by a schedule S. The precedence relation (execution order) between two operations in a schedule S is represented by $<_S$.

Of course, not all schedules are valid; only some of them preserve database consistency. For this reason, the identification of correct schedules is a crucial point in transaction management. Next, we characterize correct schedules in our model, denoted as *semantically serial*.

Definition 6. A schedule S over $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ is *semantically serial* if:
(i) it is a serial schedule or
(ii) for all $T_i \in \mathcal{T}$, T_i is *module-structured* and there is no interleaving within a module of T_i, for each module in T_i (interleavings are allowed only between each two modules). $\diamond$

Intuitively, the latter condition of Definition 6 enforces that a semantically serial schedule S, in fact, represents a serial execution of the modules, i.e., the interleaving granularity of a semantically serial schedule is a module. Example 2 shows a semantically serial schedule.

The notion of semantically serial schedules represents the first important result of our proposal. Schedules, which are non-serializable but semantically serial are correct and preserve database consistency as we prove in Theorem 1. We denote the class of *semantically serial* schedules as $S_eSerial$.

Theorem 1. A *semantically serial* schedule preserves database consistency.

Proof. Let S be a semantically serial schedule of operations performed on objects of database $\mathcal{DB}$.
Case 1. S is serial. Trivial
Case 2. S satisfies condition *(ii)* of Definition 6. Suppose S produces an inconsistent state. Without loss of generality, consider that the inconsistency results from operations on objects of semantic unit $SU_a \subseteq \mathcal{DB}$. By assumption, the execution of S represents a serial execution of modules (Definition 6.*ii*). Hence, the execution of operations on objects of SU_a in S represents a serial execution of modules of different transactions. Consequently, inconsistencies on objects of SU_a must have been produced by some module of a transaction in S (recall that there is no interleaving within a module). Thus, the inconsistency must have been caused by the execution of some transaction in S. This is obviously a contradiction (by assumption, a transaction preserves database consistency). Therefore, S cannot produce inconsistent state. ◇

Next, we give an example of a *semantically serial* schedule.

Example 2. Consider the following schedule S_c over the set $\mathcal{T}=\{T_{John}, T_{Alice}\}$ of transactions from our application example (Section 3), where:
$T_{John} = r_{John}(A)w_{John}(B)w_{John}(E)r_{John}(V)w_{John}(X)$
$T_{Alice} = r_{Alice}(E)w_{Alice}(F)r_{Alice}(P)w_{Alice}(U)w_{Alice}(V)$

$$S_c = \underbrace{r_{Alice}(E)w_{Alice}(F)}\overbrace{r_{John}(A)w_{John}(B)w_{John}(E)}\underbrace{r_{John}(V)w_{John}(X)}\overbrace{r_{Alice}(P)w_{Alice}(U)w_{Alice}(V)}$$

◇

T_{John} and T_{Alice} are module-structured transactions. Moreover, the modules of T_{John} and T_{Alice} occur in S_c without interleavings. Consequently, S_c is *semantically serial*. Observe that S_c is neither serial nor serializable with respect to the conventional model, but nevertheless S_c preserves database consistency as proved in Theorem 1.

So far, we have considered only semantically serial schedules as being “safe”. However, there are schedules which are not semantically serial, but yield the same result as a semantically serial one, that is, they may be considered as being “safe”, too. In the next section, we will define this new class of safe schedules. However, before passing on to this definition, we need to introduce two basic concepts to this definition: The concepts of *equivalent* and *semantically identical* schedules.

The concept of equivalent schedules is based on the notion of *conflict*. Two operations of different transactions *conflict*, iff they access the same database object and at least one of them is a write operation.

Definition 7. Two schedules S_1 and S_2 over the set $\mathcal{T}=\{T_1, T_2, \cdots, T_n\}$ are *equivalent*, denoted $S_1 \approx S_2$, if for any conflicting operations $p \in OP(T_i)$ and $q \in OP(T_j)$, the following condition holds: if $p <_{S_1} q$, then $p <_{S_2} q$. ◇

As already said, the notion of semantically identical transactions guarantees that different transactions yield the same result. Now we want to extend this notion in order to apply it to schedules. This will allow us to identify schedules which do not contain module-structured transactions, but produce the same result as schedules which consist of only module-structured transactions.

Definition 8. Let $\mathcal{DB}= \bigcup_{i=1}^{n} SU_i$ be a database, where $SU_i, 0 < i \leq n$, are semantic units in $\mathcal{DB}$. Two schedules S_1 and S_2 which are executed on objects of $\mathcal{DB}$ are *semantically identical*, denoted $S_1 \equiv S_2$, $:\Leftrightarrow$
(i) $OP(S_1) = OP(S_2)$ and
(ii) $(\forall p, q \in OP(S_1), x, y \in SU_k, 0 < k \leq n)(p(x) <_{S_1} q(y)) \Rightarrow (p(x) <_{S_2} q(y))$ ◇

Intuitively, the notion of semantically identical schedules considers schedules which produce the same database state, when they are executed on the same initial state. In order to illustrate this fact, consider the following example.

Example 3. Consider the schedule S_c of Example 2 and the following schedule S over $\mathcal{T}' = \{T'_{John}, T'_{Alice}\}$, where:
$T'_{John} = r_{John}(V)w_{John}(X)r_{John}(A)w_{John}(B)w_{John}(E)$
$T'_{Alice} = r_{Alice}(E)r_{Alice}(P)w_{Alice}(F)w_{Alice}(U)w_{Alice}(V)$

$S = r_{John}(V)r_{Alice}(E)w_{John}(X)r_{Alice}(P)w_{Alice}(F)w_{Alice}(U)r_{John}(A)w_{Alice}(V)w_{John}(B)w_{John}(E)$

The schedules S_c (see Example 2) and S encompass the same set of operations and order the operations on objects of the same semantic unit in the same way. Hence, they may be classified as being semantically identical, i.e., $S_c \equiv S$. Obviously, S_c and S always produce the same database state, when they are executed on the same initial state, as we can see, if we apply the following commutation processes on the schedule S:

(1) Semantic commutativity among operations of the same transaction (for example, we may commute the operation $r_{Alice}(P)$ with $w_{Alice}(F)$), and
(2) "Syntactic" commutativity among non-conflicting operations of different transactions (for example, we may commute $r_{John}(A)$ with $r_{Alice}(P)$ and $w_{Alice}(U)$).

As a result of these commutation processes, we can obtain the schedule S_c. Note, the transactions in the schedule S_c may be considered as the module-structured representation of the transactions in S, which produce the same result as T'_{John} and T'_{Alice}.

◇

It is important to notice that two semantically identical schedules order the conflicting operations in the same way. This is a consequence of the fact that only operations on objects of the same semantic unit may conflict and these operations are ordered in the same way in semantically identical schedules. However, two semantically identical schedules may not be considered equivalent, because they may be defined over different sets of transactions.

4.4 Semantically Serializable Schedules

Now, we can define a class of schedules which are not semantically serial, but do not violate consistency because they have the same behavior as some semantically serial schedules.

Definition 9. Let S_{ss} be a semantically serial schedule. A schedule S is *semantically serializable* if:
(i) $S \approx S_{ss}$ or
(ii) there is a schedule S', such that $S \equiv S'$ and $S' \approx S_{ss}$ ◇

There may be some schedules which are not equivalent to a semantically serial schedule but which produce the same state as a schedule S' (because they are *semantically identical*), where S' is *equivalent* to a semantically serial schedule. Those schedules can also be considered as being "safe" as expressed by condition *(ii)* of Definition 9. With $\mathrm{S_eSR}$ we denote the class of semantically serializable schedules.

Example 4. Consider the following schedule S over $\mathcal{T}' = \{T'_{John}, T'_{Alice}\}$, where:
$T'_{John} = r_{John}(V)w_{John}(X)r_{John}(A)w_{John}(B)w_{John}(E)$

$T'_{Alice} = r_{Alice}(E) r_{Alice}(P) w_{Alice}(F) w_{Alice}(U) w_{Alice}(V)$
$S = r_{John}(V) r_{Alice}(E) w_{John}(X) r_{Alice}(P) w_{Alice}(F) w_{Alice}(U) r_{John}(A) w_{Alice}(V) w_{John}(B) w_{John}(E)$

The transaction T'_{Alice} in S is not module-structured. For this reason, there is no semantically serial schedule which is equivalent to S (remember that semantically serial schedules only consist of module-structured transactions). Now, consider the following schedule S':
$S' = r_{John}(V) r_{Alice}(E) w_{Alice}(F) w_{John}(X) r_{Alice}(P) w_{Alice}(U) r_{John}(A) w_{Alice}(V) w_{John}(B) w_{John}(E)$

Observe that $S \equiv S'$. Furthermore, S' is equivalent to the schedule S_c of Example 2 which is semantically serial. Consequently, the schedule S is semantically serializable. $\diamond$

Definition 10. Let S be a schedule over $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$. The *Semantic Serialization Graph* for S is a directed graph $S_eSG(S) = (N, E)$ in which $N = \mathcal{T}$. E represents the set of labeled edges $T_i \xrightarrow{SU_n} T_j$, where $T_i, T_j \in N$ and there are two operations $p \in OP(T_i), q \in OP(T_j)$, $p <_S q$, on an object of the *semantic unit* SU_n, which are in conflict.

Theorem 2. A schedule S is semantically serializable iff $S_eSG(S)$ contains no cycle over edges with the same label.

Proof. See Appendix. $\diamond$

In Figure 2, we give an example of the semantical serialization graph resulting from the execution of the schedule S shown in Example 4 which is semantically serializable.

It is easy to note that checking, whether a semantical serialization graph of a schedule S is acyclic or not, can be done in $O(m.n^2)$ time, where m is number of labels and n the number of transactions in S. Thus, the semantical serialization graph provides an *efficient* method to identify semantically serializable schedules.

Thus far, we have analyzed the correctness of complete schedules which are represented by complete executions of transactions. However, in a dynamic context it is important that a correctness criterion is *prefix-closed* [4], that is, if a schedule S is correct, then any prefix of S is also correct. Next, we show that the notion of semantic serializability captures this property.

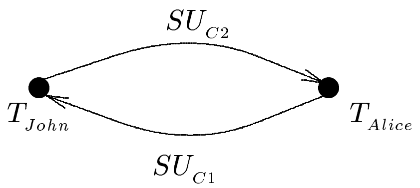

Figure 2: Semantical serialization graph of the schedule S (Example 3).

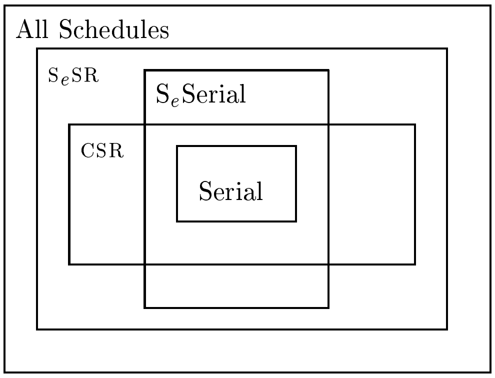

Figure 3: Relationships among different correctness classes.

Theorem 3. Semantic serializability is a prefix-closed schedule property.

Proof. Let S be a semantically serializable schedule and S' an arbitrary prefix of S. By Theorem 2, $S_eSG(S)$ is acyclic. Evidently, $S_eSG(S')$ is a subgraph of $S_eSG(S)$. Consequently, $S_eSG(S')$ is acyclic, too. By Theorem 2, S' is semantically serializable. $\diamond$

It is important to note that not all correctness criteria for schedules are prefix-closed. For example, final state and view state serializability are not prefix-closed [15].

The Venn diagram shown in Fig. 3 illustrates the relationships among the classes of schedules of our model and the classes CSR (conflict serializable [4, 9]) and *Serial* of the conventional transaction model.

It is easy to verify that the class CSR is a subset of the class $\mathrm{S}_e\mathrm{SR}$. We do this by considering a schedule $S' \in$ CSR. Thus, S' must be equivalent to a serial schedule [9]. By Definition 9, S' is also semantically serializable, because S' is equivalent to a serial schedule and every serial schedule is semantically serial (by Def. 6). On the other hand, the schedule S illustrated in Example 4 is semantically serializable, but not conflict serializable. Since S' is an arbitrary schedule of the class CSR and the schedule S of Example 4 is not in class CSR, the relation CSR $\subseteq \mathrm{S}_e\mathrm{SR}$ follows.

To verify that CSR$\backslash S_eSerial \neq \emptyset$ and $S_eSerial\backslash$ CSR$\neq \emptyset$, consider the following schedules:
$S_1 = r_{Alice}(E)w_{John}(E)w_{Alice}(F)$
$S_2 = r_{Alice}(E)r_{John}(V)w_{John}(E)w_{Alice}(V)$

The schedule S_1 is conflict serializable, but not $S_eSerial$. On the other hand, the schedule S_2 is not conflict serializable, but it is $S_eSerial$.

5 Concurrency Control Mechanism

In this section, we present two protocols which may be used by a scheduler to produce semantically serializable schedules. One protocol is based on a locking mechanism and the other one on the enforcement of the acyclicity of a semantic serialization graph.

5.1 The Semantic Locking Protocol

The Semantic Locking Protocol, denoted $\mathrm{S}_e\mathrm{L}$, is based on the concepts mentioned in the previous sections. As we will see, the $\mathrm{S}_e\mathrm{L}$ protocol may increase the concurrency among transactions and may optimize the processing of long transactions by reducing the frequency of deadlocks. Those additional benefits of an $\mathrm{S}_e\mathrm{L}$ protocol can especially be achieved if there is some interaction between the user or transaction programmer and the concurrency control mechanism. For that reason, we make the following assumptions:

(a) The user may mark the last operation performed in a transaction on objects of a given semantic unit. In comparison to 2PL, this information enables the scheduler to increase concurrency.
(b) The user may specify if a read operation on an object is a read-only operation or a read with a future intension to update it. This information reduces the frequency of deadlocks resulting from lock conversions [19].

In the case of interactive transactions, the last operation on a semantic unit (assumption (a) described above) can be marked *ad hoc* by the user. Such a feature makes the $\mathrm{S}_e\mathrm{L}$ protocol able to control concurrency of applications whose transactions have an interactive nature, e.g. CAD and CASE applications. Observe that the chopping procedure proposed in [24] must be performed before transactions are executed. Thus, it can not be applied to transactions in such interactive applications.

The $\mathrm{S}_e\mathrm{L}$ protocol utilizes three types of locks: read-only, write and update. A *read-only lock*, denoted *rl*, only allows a transaction to read a database object, but, after a read-only lock is acquired, the transaction may not obtain a write lock on the same object. Therefore, the $\mathrm{S}_e\mathrm{L}$ protocol does not allow a lock conversion from a read-only lock to a write lock. In order to support that an object may be read and written in a transaction, we provide the update lock [19]. An *update lock* (*upl*) represents a read lock which may be converted to a write. A *write lock* grants exclusive access. All lock operations are released with *unlock* operations, one for each lock type: *rul*, *wul* and *upul*.

It is important to note that a transaction with an update lock on an object x must request and, evidently, acquire a write lock on x in order to execute a write operation on x.

Two lock types are defined as *compatible* if they may be held simultaneously by different transactions on the same object. Compatibility in $\mathrm{S_eL}$ is shown in Figure 4. Since a read-only lock captures the information that an object will be used only for a read operation without the intension to update it, update locks are compatible with read-only locks. This allows a higher degree of cooperation among long transactions and represents a deviation from the original proposal of update lock as presented in [19]. On the other hand, update locks are not compatible with themselves and with write locks. For this reason, we call update locks *partially exclusive*.

In our approach, a transaction may consist of several atomic units, each of them representing a sequence of operations on objects of exactly one semantic unit. This allows us to design the locking mechanism as follows.

First, we may introduce the two-phase property of the conventional 2PL protocol w.r.t. an *atomic unit* of a transaction and not w.r.t. the entire transaction. This way we achieve more cooperation among transactions. For each atomic unit there is a *growing phase* in which locks may be acquired and a *shrinking phase* in which locks are released. The shrinking phase may not start before the last operation of the atomic unit has been executed.

Second, we may introduce aspects of a *variable granularity locking* strategy [14]. Granularity refers to the size of the objects involved in a lock. The idea is to give the user the option to choose the "optimal" lock granularity for each atomic unit of a transaction. Concurrency may be enhanced by fine granularity, whereas locking overhead may be reduced by coarse granularity.

The $\mathrm{S_eL}$ protocol supports three lock granules: *database*, *semantic unit* and *object*. The relationship among these granules may be represented by means of the following lock graph: *database*$\longrightarrow$*semantic unit*$\longrightarrow$*object*. Each node of this graph may be locked. A lock on a granule *implicitly* locks all its descendants (if there is some) and propagates the effect of the locking activity to all its ancestors as *intension* lock. Thus, for each lock type we additionally define an *intention* lock (irl, iwl, iul).

	rl_i	irl_i	wl_i	iwl_i	upl_i	$iupl_i$	$\mathcal{N}il$
rl_j	+	+	-	-	+	+	+
irl_j	+	+	-	+	+	+	+
wl_j	-	-	-	-	-	-	+
iwl_j	-	+	-	+	-	+	+
upl_j	+	+	-	-	-	-	+
$iupl_j$	+	+	-	+	-	+	+

Figure 4: Compatibility matrix.

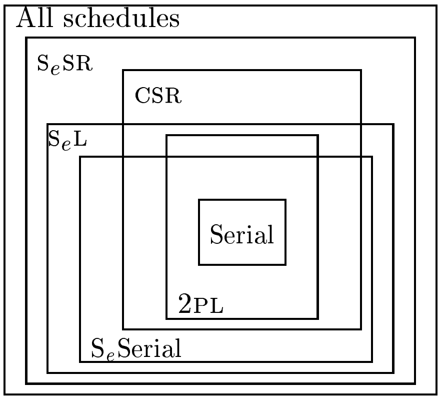

Figure 5: Relationship among classes of correct schedules.

In the compatibility matrix of Fig. 4, the type $\mathcal{N}il$ represents the absence of locks on x. All the locks in the $\mathrm{S_eL}$ protocol are set according to the compatibility matrix.

For each atomic unit $\Pi_{[OP_{SU_i}(T)]}(T)$ of a transaction T, a S_eL scheduler sets and releases locks on any node x of the lock graph according to the following rules:

(R1) To set an $rl(x)$, the transaction T must first set irl on all ancestor nodes of x according to the lock compatibility shown in Fig. 4. If T already holds iwl or iul on any ancestor of x, these locks remain unchanged (iwl and $iupl$ are stronger than irl);

(R2) To set a $wl(x)$, the transaction T must first set iwl on all ancestor nodes of x;

(R3) To set a $upl(x)$, the transaction T must first set $iupl$ on all ancestor nodes of x. If T already holds iwl on any ancestor of x, then the iwl remain unchanged (iwl is stronger than $iupl$);

(R4) An intensional lock on x may not be released by T, if it holds a lock on any descendants of x;

(R5) The locks held by T on objects of a semantical unit SU_i may be released after the last operation of $\Pi_{[OP_{SU_i}(T)]}(T)$.

Theorem 4. Let S_eL be the set of schedules produced by a S_eL scheduler. Then $\text{S}_e\text{L} \subset \text{S}_e\text{SR}$.

Proof. Let S be a schedule produced by a S_eL protocol, that is, $S \in \text{S}_e\text{L}$. Suppose, by way of contradiction, that $S \notin \text{S}_e\text{SR}$. By Theorem 2, if $S \notin \text{S}_e\text{SR}$, then the semantic serialization graph of S is cyclic. Without loss of generality, consider that the cycle in $S_eSG(S)$ is over edges with label SU_n and has the following form: $T_i \xrightarrow{SU_n} T_j \xrightarrow{SU_n} T_i$. It follows from this that, for some operations $p_i(x), q_i(y), x, y \in SU_n$, the transaction T_i may obtain a lock on y after having released a lock on x, a contradiction to rule R5. Thus, $\text{S}_e\text{L} \subseteq \text{S}_e\text{SR}$. Consider the following schedule over transactions of our application example: $S' = r_{Alice}(E)w_{John}(E)w_{Alice}(F)$ (recall that E,F are objects of the semantic unit SU_{C1}). Clearly, $S' \in \text{S}_e\text{SR} \backslash \text{S}_e\text{L}$. Thus, the theorem follows. $\diamond$

Theorem 4 guarantees that an S_eL protocol will produce only semantically serializable schedules. Fig. 5 illustrates the relationship among the different classes of schedules. Observe that the class of schedules produced by a 2PL protocol is a subset of the class S_eL. In 2PL, the locks are released when a transaction commits (or aborts). Thus, the 2PL rules are a special case of the S_eL rules. Consequently, any schedule which is in the class 2PL is also in S_eL. The schedule shown in Example 4 is in S_eL, but not in 2PL. Thus, $\text{2PL} \subset \text{S}_e\text{L}$. This relationship means that an S_eL protocol supports a higher degree of concurrency than a 2PL protocol. Thus, a concurrency control mechanism based on our proposal allows more cooperation among design activities in a CAD environment.

5.2 The S_eSG Checking Protocol

In Section 5.1, we introduce a protocol for concurrency control which produces a subset of all possible correct schedules, i.e., a subset of the class S_eSR. In this section, we describe another protocol which can produce all possible semantically serializable schedules over a given set of transactions. The protocol, denoted S_eSGC, is based on a similar strategy which is used by the conventional serialization graph testing protocol: *the dynamic monitoring and management of a conflict graph, which should always remain acyclic*. However, an S_eSGC protocol additionally exploits semantic knowledge provided by the notion of semantic units.

The conflict graph, denoted $\mathcal{C}$-graph, maintained by an S_eSGC protocol is constructed according to the same rules used to construct a semantic serialization graph (Def. 10). However, a $\mathcal{C}$-graph differs from an S_eSG in two aspects. First, not all committed transactions must be represented. Second, not all conflicts must be represented as edges. In some cases, edges may be "safely" discarded. Later we will show how this can be done.

Next, we describe how an S_eSGC protocol manages the $\mathcal{C}$-graph. As an S_eSGC scheduler starts running, the $\mathcal{C}$-graph is created as an empty graph. As soon as the scheduler receives the first operation of a new transaction (*begin-transaction*) T_i, a node representing this transaction is inserted in the $\mathcal{C}$-graph. For each operation $p_i(x) \in OP(T_i)$ which is received, the scheduler checks if there is a conflicting operation $q_j(x) \in OP(T_j)$ which was already scheduled. If an operation $q_j(x)$ was already scheduled, the scheduler inserts an edge of the form $T_j \xrightarrow{SU_n} T_i$, where x is an object of the semantic unit SU_n. Now, the scheduler verifies if the new edge introduces a cycle in the $\mathcal{C}$-graph. In the affirmative case, the scheduler rejects $p_i(x)$, undoes the effect of operations of the atomic unit $\Pi_{[OP_{SU_n}(T_i)]}(T_i)$ and removes the edge $T_j \xrightarrow{SU_n} T_i$ from $\mathcal{C}$-graph. Otherwise, $p_i(x)$ is accepted and scheduled.

Note, if the scheduler identifies a cycle in the $\mathcal{C}$-graph, the S_eSGC protocol requires that only the operations of the atomic unit which contains the operation causing the cycle are rolled back. It is not necessary to abort the entire transaction. This is an important property for applications with long-living transactions.

Theorem 5. Let S_eSGC be the set of schedules produced by an S_eSGC scheduler. Then $\text{S}_e\text{SGC} = \text{S}_e\text{SR}$.
Sketch of Proof. It is easy to show that $\text{S}_e\text{SGC} \subset \text{S}_e\text{SR}$. We have only to observe that every schedule S which is produced by an S_eSGC scheduler has an acyclic $\mathcal{C}$-graph. It follows that $S \in \text{S}_e\text{SR}$, thereby $\text{S}_e\text{SGC} \subset \text{S}_e\text{SR}$. To verify that $\text{S}_e\text{SGC} \supset \text{S}_e\text{SR}$, we have to show that every schedule $S \in \text{S}_e\text{SR}$ may be the result of a S_eSGC-scheduler. We show this by induction on the length of S that every operation p in S may not originate a cycle in the $\mathcal{C}$-graph (recall that S is semantically serializable) and, thus, p may be executed. $\diamond$

Obviously, the maintenance of a $\mathcal{C}$-graph consumes a lot of memory space. In order to minimize this problem, we propose a garbage collection algorithm whose main functionality is to safely remove nodes and edges from that graph. The garbage collection on the $\mathcal{C}$-graph$= (N, E)$ is based on the following rules:

(R1) For each $T_i \in N$ (recall that each transaction T_i is represented by a node in the $\mathcal{C}$-graph) and for each atomic unit $\Pi_{[OP_{SU_n}(T_i)]}(T_i)$ of T_i: If the last operation of $\Pi_{[OP_{SU_n}(T_i)]}(T_i)$ has already been scheduled and the node T_i is a source[1] w.r.t. edges with label SU_n , then edges with origin in T_i may be removed from the $\mathcal{C}$-graph;

(R2) If a node representing a committed transaction T_i is disconnected from the $\mathcal{C}$-graph (i.e., the node is unreachable), it may be removed.

In order to show that the rule R1 does not jeopardize the correctness of an S_eSGC scheduler, consider the following scenario. The last operation of an atomic unit $\Pi_{[OP_{SU_n}(T_i)]}(T_i)$ has already been executed. From that point on, no edge with label SU_n may point to T_i, because, by Def. 4, an atomic unit contains all operations of T_i to be performed on objects of a semantic unit. Thus, if there is no more edge with label SU_n arriving in T_i, then T_i may not be involved any more in a cycle over edges with label SU_n. Therefore, the edges starting from T_i and having the same label may be removed.

The application of rule R2 is a consequence of the first rule. Therefore, executing the garbage collection procedure does not affect the correctness of an S_eSGC protocol.

[1] A node represents a *source* w.r.t edges with label SU_n in a directed graph, if it has only emanating edges with label SU_n, that is, it has no incoming edge with label SU_n.

6 Discussion

Some issues related to our proposal which were not discussed in this paper are the *modularization of the concurrency control mechanism*, user-controlled interruption/continuation of transaction execution and *transaction failure*.

Semantic serializability supports the modularization of the concurrency control mechanism. The idea here is to distribute the synchronization function of a concurrency control mechanism across several schedulers. This can be realized by implementing a concurrency control mechanism which consists of several schedulers (instead of only one), where the basic function of each scheduler is to synchronize the operations on objects of only one semantic unit. The schedulers may be located on different sites, may be based on different protocols, and they may run simultaneously. For example, in a scenario like our application example (see Sec. 3), our model supports the following transactional approach. Two schedulers may be defined to simultaneously control the database operations: Scheduler A assigned to schedule operations on objects of a semantic unit SU_{C1} is located on a site A and uses an $\mathtt{S}_e\mathtt{L}$ protocol. On the other hand, scheduler B, responsible for scheduling operations on objects of a semantic unit SU_{C2}, may be located on a site B and may implement an $\mathtt{S}_e\mathtt{SGC}$ protocol. Evidently, by the modularization of the concurrency control mechanism, more concurrency may be achieved.

The notion of user-controlled interruption/continuation of transaction execution proposed in [18] may be supported by our approach. In our proposal, the user may interact with the concurrency control mechanism after the last operation of each atomic unit of a transaction in order to suspend the execution of a design activity (i.e., a transaction) and, after a time interval, to resume its execution.

To cope with transaction failures, the notion of savepoints [18] may be used. The idea is to define savepoints after the last operation of each atomic unit of a transaction. Hence, after a transaction failure, the transaction is rolled back to the last savepoint and is resumed from that point on. Another approach is to use the notion of compensating functions [11]. For this approach, a kind of "semantic" countersteps should be defined for each atomic unit of a transaction. We are currently investigating mechanisms to cope with system and media failures [16].

We have shown throughout this paper the applicability of *semantic serializability* to a wide variety of advanced database applications. More specifically, we identify applications which have long-living transactions and requires a cooperative environment, such as CAx and WFMS, as applications for which *semantic serializability* is especially useful.

In order to verify the applicability of our proposal in a *workflow* application, consider the scenario illustrated in Figure 6. A_i and A_j represent activities of a workflow which should be executed concurrently. The actions of activity A_i access the database objects X and Y and the actions of A_j the object X. Assume that X and Y present no *depends on* relation (see Section 4). Suppose that A_i has already started and is executing operations on X and Y. If the operations on X and Y are scheduled by a scheduler which implements a 2PL protocol, the activity A_j may only begin its execution after A_i has terminated. Thus, A_i and A_j may not be executed concurrently, although, according to the workflow specification, they should be executed simultaneously. However, a scheduler based on our approach allows that A_i, A_j are really executed concurrently.

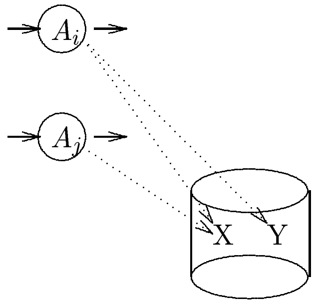

Figure 6: Scenario for a workflow application.

However, the applicability of our proposal is not restricted to applications in CAx or WFMS environments. We identify other classes of database applications for which our model is useful, for example, multidatabase systems, temporal databases and GIS. Next, we describe some application scenarios in order to show that our transaction mechanism may be used to optimize the transaction processing of those advanced database applications.

A *multidatabase system* (MDBS) integrates a collection of "pre-existing" local databases. Such local databases were created independently and in an uncoordinated way without considering that they will be integrated sometime in the future [6]. Hence, it is reasonable to see a multidatabase as a collection of disjoint groups of objects, each of which representing a local database where there is no *depends on* relation, i.e., do not exist update dependencies (see Section 4) among objects of different groups. For that reason, each local database may be defined as a *semantic unit* for the global scheduler. Evidently, a global transaction may have operations on different semantic units (local databases). Figure 2 illustrates a multidatabase application. This application represents the implementation of the project example described in Section 3 in a MDBS. Observe that the design objects of each circuit (see Figure 1) are stored in CAD databases (LDB_1 and LDB_2) located at different sites. The two databases are managed by heterogeneous DBMSs. Notwithstanding, designers carry out the project at a third site, where the CAD tools are installed. It should be transparent for the designers that the design objects are stored in multiple heterogeneous database systems.

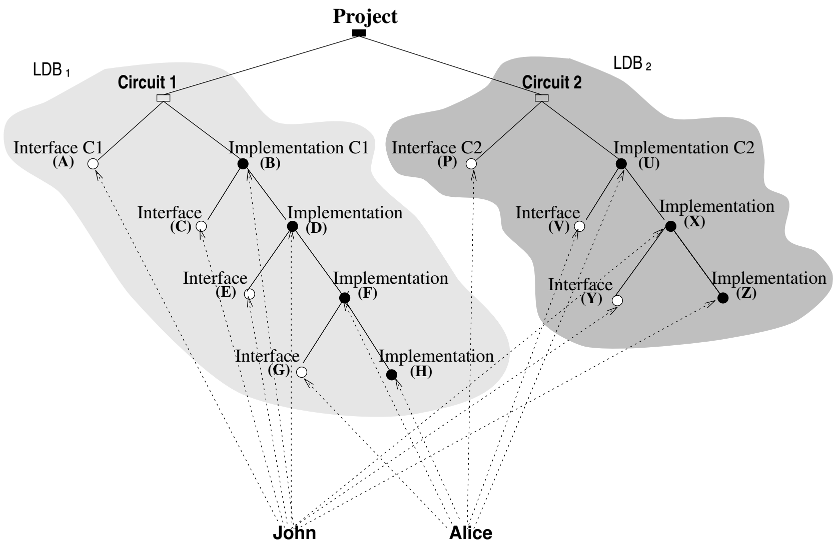

Figure 2: Example of a multidatabase application.

As shown before, semantic serializability preserves database consistency. Hence, multidatabase transactions do not violate multidatabase consistency if they are synchronized by an MDBS implementing semantic serializability and are executed alone at the local systems. However, global transactions do not run alone on each local database. There are local transactions running outside the control of the MDBS. Such a context could lead transactions to behave incorrectly, since the MDBS has no information about local transactions. Fortunately, we can assume that each local DBMS ensures database consistency. Thus, *semantic serializability* may use the following criterion *additionally* to enforce correctness of global schedules (recall that a global schedule consists of operations belonging to local as well as to global transactions): *schedules on each local database system must be correct.* Thus, provided that local DBMSs assure the correctness of local schedules and the MDBS enforces semantic serializability for the execution of global transactions, the global schedule is correct, that is, it guarantees multidatabase consistency. Since existing DBMSs use "syntactical" serializability as correctness criterion, we can consider that local schedules are correct if they are serializable. This model for processing transactions in a multidatabase context contrasts with the idea of *2LSR* [6] and quasi serializability [8], which requires serializable execution of global transactions. In contrast to *2LSR*, a global scheduler implementing *semantic serializability* relaxes serializability and enforces database consistency. Consequently, global schedulers based on our approach allow more concurrency among global transactions.

The notion of semantic unit is flexible enough to allow that two or more local databases can be logically grouped in order to represent a single semantic unit, without violating local autonomy of each local system. By "logically", we mean that only the MDBS should be aware of such a representation. It is not necessary to physically join neither the databases nor the database systems.

Specifying more than one local database as a single semantic unit enables our transaction-processing model to synchronize transactions in MDBSs which have the following characteristics:

- Data replication. Some MDBSs may contain objects replicated in more than one local database. In this case, the local databases containing replicated data should be grouped within a single semantic unit

- Global constraints. Constraints spanning several local database are called global constraints. In other words, global constraints involve objects of more than one local database. For some MDBSs, global constraints are specified. Local databases containing objects referred in a global constraint should determine a semantic unit.

In applications using *temporal databases* [5, 25], each time slice for *rollback-time* represents a snapshot of the database, i.e., a database state in a given time point (or time interval). Thus, different slices represent different database states. The values of objects in each slice for *rollback-time* may only be updated based on values of the same rollback-time. Hence, time slices w.r.t. rollback-time may be defined as semantic units. Thus, the values of objects belonging to different rollback-times will be represented in different semantic units. Transactions in a temporal database may execute operations on objects whose values are associated with different rollback-times, and are, therefore in different semantic units. Hence, *semantic serializability* may be used for the synchronization of transactions in temporal databases allowing for more concurrency among them.

In a *GIS* [7, 28], the underlying spatial database may contain regions which have a *disjoint* topological relationship. Such regions are spatially disjoint. It is easy to observe that spatial objects belonging to disjoint regions do not present any *depends on* relation. That is, an update operation on an object of a region R_A does not depend on values of objects belonging to regions spatially disjoint from R_A, which are read by the same transaction. For that reason, the regions which are spatially disjoint may be considered as semantic units. Transactions in a GIS may access objects belonging to different spatial regions; most of them typically last very long. Therefore, a transaction mechanism implementing *semantic serializability* may increase the concurrency among transactions for a GIS environment. However, the specification of semantic units in a GIS environment may be application specific. For that reason, we are currently investigating the definition of a hierarchy of semantic units in order to overcome this problem. The basic idea is to allow that a semantic unit can be defined as an aggregation of two or more semantic units. For instance, a semantic unit SU_1 can be defined as the union of the semantic units SU_{1_1} and SU_{1_2}. Such a hierarchy supports that different geographic applications have distinct "views" of the semantic units. For example, a geographic application A may view the spatial database as a collection of the following semantic units: SU_{1_1}, SU_{1_2} and SU_2. On the other hand, an application B views the same database as a collection of the semantic units SU_1 and SU_2.

7 Conclusions

In this paper, we have proposed a new approach to increase the parallelism of transaction processing by relaxing the notion of atomicity of transactions. A theory for the processing of concurrent transactions is presented which is based on the use of semantic knowledge. This theory introduces a new concept for correctness of schedules, denoted *semantic serializability*, as well as a new notion of equivalence for schedules. We have shown that the class of semantically serializable schedules is larger than the class of correct schedules in the conventional model. Semantically serializable schedules preserve database consistency and may be recognized efficiently (by checking the acyclicity of a direct graph). *Semantic serializability* captures the prefix-closed property for schedules.

Moreover, we have developed two concurrency control protocols to obtain semantically serializable schedules. One protocol is based on a locking mechanism whereas the other one uses a non-locking approach.

We conclude emphasizing the applicability of *Semantic Serializability* for a wide variety of advanced database applications.

Acknowledgments

We would like to thank the anonymous referees for sugestions to improve the presentation.

A Appendix

Lemma 1. Let S be a schedule over a set $\mathcal{T}$ of transactions and $\mathcal{M}_S(SU_a)$ be the subsequence of operations in S executed on objects of a semantic unit SU_a, such that, for each $p_i \in OP(T_i)$ and $q_j \in OP(T_j), T_i, T_j \in \mathcal{T}$, p_i and q_j in $\mathcal{M}_S(SU_a)$, if $p_i <_S q_j$, then $p_i <_{\mathcal{M}_S(SU_a)} q_j$. Furthermore, let S_{SS} be a semantically serial schedule over $\mathcal{T}$. For each $\mathcal{M}_S(SU_\alpha) \neq \emptyset$ in S, with $\alpha = a, b, c, \ldots$, where each SU_α is a semantic unit, if $\mathcal{M}_S(SU_\alpha) \approx \mathcal{M}_{S_{SS}}(SU_\alpha)$, then $S \in$ S$_e$SR.

Proof. Let $\mathcal{M}_S(SU_\alpha) \approx \mathcal{M}_{S_{SS}}(SU_\alpha)$, $\forall \mathcal{M}_S(SU_\alpha) \neq \emptyset$, with $\alpha = a, b, c, \ldots$.
Assume for a semantic unit SU_a,$\mathcal{M}_S(SU_a) \neq \emptyset$. By assumption, we have $\mathcal{M}_S(SU_a) \approx \mathcal{M}_{S_{SS}}(SU_a)$. That is, $\mathcal{M}_S(SU_a)$ and $\mathcal{M}_{S_{SS}}(SU_a)$ order the conflicting operations in the same way. Thus, for each pair of operations $p_i \in OP(T_i)$ and $q_j \in OP(T_j)$, $T_i, T_j \in \mathcal{T}$, which are executed on the same object of the semantic unit SU_a, if p_i occurs before q_j in $\mathcal{M}_S(SU_a)$, then p_i appears before q_j in $\mathcal{M}_{S_{SS}}(SU_a)$, too. Suppose, by way of contradiction, that $S \notin$ S$_e$SR. Without loss of generality, consider that $S \notin$ S$_e$SR because no semantically serial schedule over $\mathcal{T}$ orders the operations $p_i(x)$ and $q_j(x)$, $x \in SU_a$ in the same way as S. Thus, if $p_i(x) <_S q_j(x)$, then $q_j(x)$ occurs before $p_i(x)$

in all semantically serial schedules over $\mathcal{T}$, a contradiction, since $\mathcal{M}_S(SU_a) \approx \mathcal{M}_{S_{SS}}(SU_a)$ by our assumption, where S_{SS} is a semantically serial schedule over $\mathcal{T}$. Therefore, $S \in$ S$_e$SR. $\diamond$

Lemma 2. If a schedule S is semantically serial, then $S_eSG(S)$ is acyclic.

Proof. Let $S \in$ S$_e$Serial be a schedule over $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$.
Case 1. S is serial. Trivial.
Case 2. S satisfies Condition 6.*ii* (Def. 6).
By Condition 6.*ii*, S represents a serial execution of modules of each transaction $T_i \in \mathcal{T}$, $0 < i \leq n$. Suppose, by way of contradiction, that $S_eSG(S)$ is cyclic and, without loss of generality, the cycle has the following form: $T_i \xrightarrow{SU_n} T_j \xrightarrow{SU_n} T_i$. It follows from this that the *module* of T_i (Def. 5) which represents the operations of T_i on objects of the semantic unit SU_n is interleaved by some operations of T_j, a contradiction, by assumption, S satisfies Condition 6.*ii*. Thus, the lemma follows. $\diamond$

Theorem 2. A schedule S is semantically serializable iff $S_eSG(S)$ contains no cycle over edges with the same label.

Proof.
"$\Rightarrow$": Let S be a semantically serializable schedule.
Case 1. $S \approx S_{SS}$, where S and S_{SS} are schedules over the set $\mathcal{T}$ of transactions and $S_{SS} \in$ S$_e$Serial. Suppose, by way of contradiction, that $S_eSG(S)$ is cyclic. Without loss of generality, consider that $S_eSG(S)$ contains the following cycle: $T_i \xrightarrow{SU_n} T_j \xrightarrow{SU_n} T_i$. By assumption, S and S_{SS} are equivalent. It follows from this that $S_eSG(S_{SS})$ has the same cycle as $S_eSG(S)$, a contradiction to Lemma 2 since S_{SS} is semantically serial. Therefore, $S_eSG(S)$ is acyclic.
Case 2. $S \equiv S'$ and $S' \approx S_{SS}$, where $S_{SS} \in$ S$_e$Serial.
$S' \approx S_{SS}$, thus $S_eSG(S')$ is acyclic (by proof for *Case 1*). By assumption, $S \equiv S'$. Thus, S and S' order the operations on objects of a semantic unit in the same way (by Def. 8). It follows from this that S and S' order the conflicting operations in the same way. Thus, $S_eSG(S)$ is acyclic, too.
"$\Leftarrow$": $S_eSG(S)$ contains no cycle over edges with the same label.
Consider that $S_eSG(S)$ contains edges with label SU_n. Since $S_eSG(S)$ is acyclic, we may topologically sort it by edges with label SU_n. However, for this topological sort, instead of considering a transaction T_i as a node, we consider only the atomic unit $\Pi_{[OP_{SU_n}(T_i)]}(T_i)$. Thus, as result of this "modified" topological sort, we obtain a serial execution of atomic units of transactions in S which contains operations on objects of the semantic unit SU_n. In fact, this serial execution of atomic units represents a serial execution of modules (see Def. 5). If we repeat "recursively" this process for each label in $S_eSG(S)$, we obtain a serial execution of modules of all transactions in S. Moreover, there is no interleaving within each module. We have, thus, a semantically serial schedule (by Def. 6) which we call S_{SS}. It is easy to see that, for each $\mathcal{M}_S(SU_\alpha) \neq \emptyset$ (see Lemma 1), $\mathcal{M}_S(SU_\alpha) \approx \mathcal{M}_{S_{SS}}(SU_\alpha)$. Thus, by Lemma 1, S is semantically serializable. The theorem follows. $\diamond$

References

[1] Agrawal, D., Bruno, J.L., Abbadi, A. El and Krishnaswamy, V. Relative Serializability: An Approach for Relaxing the Atomicity of Transactions. In *Proceedings of the ACM Symposium on PODS*, pages 139–149, 1994.

[2] Bancilhon, F., Kim, W. and Korth, H. F. A Model of CAD Transactions. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 25–33, Stockholm, August 1985.

[3] Batory, D. S. and Kim, W. Modeling Concepts for VLSI CAD Objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[4] Bernstein, P. A., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] Brayner, A. and Bauzer Medeiros, C. Incorporating Temporal Dimensions in a Object-Oriented DBMS. In *Proccedings of the 9th. Brazilian Database Symposium (SBBD)*, pages 16–29, 1994.

[6] Breitbart, Y., Garcia-Molina, H., Silberschatz, A. Overview of multidatabase transaction management. *The VLDB Journal*, (2):181–239, 1992.

[7] E. Clementini, J. Sharma, and M. Egenhofer. Modelling Topological Spatial Relations: Strategies for Query Processing. *Computer & Graphics*, 18(6):815–822, 1994.

[8] Du, W. and Elmagarmid, A. K. Quasi Srializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the 15th International Conference on VLDB*, pages 347–355, Amsterdam, 1989.

[9] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[10] Farrag, A. A., and Özsu, M. T. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[11] Garcia-Molina, H. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[12] Garcia-Molina, H. and Salem, K. . SAGAS. In *Proceedings of the ACM SIGMOD Conference*, pages 249–259, 1987.

[13] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann, 1993.

[14] Gray, J.N., Lorie, R.A., Putzolu, G.R. and Traiger, I.L. Granularity of Locks and Degrees of Consistency in a Shared Database. In M. Stonebraker, editor, *Readings in Database Systems*, pages 94–121. Morgan-Kaufmann, 1988.

[15] Hadzilacos, V. A Theory of Reliability in Database Systems. *Journal of the ACM*, 35:121–145, 1988.

[16] Härder, T. and Reuter, A. Principles of Transaction-Oriented Database Recovery. *Computing Surveys*, 15(4):287–317, 1983. Reprinted in: Readings in Database Systems, Third Edition M. Stonebraker, J. Hellerstein (eds.), Morgan Kaufmann Publishers, March 1998.

[17] Härder, T. and Rothermel, K. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39–74, 1993.

[18] Härder, T., Hübel, Ch., Meyer-Wegener, K. and Mitschang, B. Processing and transactions concepts for cooperation of engineering workstations and a database server. *Data & Knowledge Engineering*, (3):87–107, 1988.

[19] Korth, H. F. Locking Primitives in a Database System. *Journal of the ACM*, 30(1):55–79, 1983.

[20] Lynch, N. A. Multilevel Atomicity – A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[21] Moss, J.E.B. Nested Transactions: an approach to reliable computing. Technical Report MIT-LCS-TR-260, M.I.T, Laboratory of Computer Science, 1981.

[22] Papadimitriou, C. H. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26:631–653, 1979.

[23] Papadimitriou, C. H. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[24] Shasha, D., Simon, E. and Valduirez, P. Simple Rational Guidance for Chopping Up Transactions. In *Proceedings of 1992 ACM SIGMOD Conference*, pages 298–307, 1992.

[25] Richard T. Snodgrass. Temporal Databases. In *Theories and Methods of Spatial-Temporal Reasoning in Geographic Space*, pages 22–64. Spring-Verlag, 1992. Lecture Notes in Computer Science - Vol. 639.

[26] Vidyasankar, K. Generalized Relative Serializability. Technical report, Department of Computer Science, Memorial University of Newfoundland, 1995.

[27] Weikum, G. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[28] M. F. Worboys and S. M. Deen. Semantic Heterogeneity in Distributed Geographic Databases. *SIGMOD RECORD*, 20(4):30–34, December 1991.