


Definition 8: Let $RT = e_0 : \text{reg\_exp}$ be a regular term containing only elements with arity 2 or less. Let $(G_0, s_{i_0})(s_{i_0}, G_1) \cdots (G_n, s_{i_n-1})(s_{i_n-1}, G_0)$, $n > 1$, be a strong-cycle in a TSGD. The strong-cycle $(G_0, s_{i_0}) \cdots (s_{i_n-1}, G_0)$ is said to be consistent with respect to $RT$ iff

- $e_0 = \text{type}(G_0 : G_{0 \mid i_{n-1}}, G_{0 \mid i_0})$, and
- $\text{type}(G_1 : G_{1 \mid i_0}, G_{1 \mid i_1}) \cdots \text{type}(G_{n-1} : G_{(n-1) \mid i_{n-2}}, G_{(n-1) \mid i_{n-1}})$ is a string in $L(\text{reg\_exp})$.

A TSGD is said to be strongly-acyclic with respect to a regular specification $R$ iff for every $RT \in L(\text{reg\_exp})$, it does not contain a strong-cycle consistent with respect to $RT$. □

The problem of determining if the invariant holds can be shown to be NP-complete as a consequence of the following NP-completeness result.

**Theorem 8:** The following problem is NP-complete: Given a TSGD $(V, E, D, L)$, such that it is consistent, a regular specification $R$ containing only elements with arity 2, does there exist a set of dependencies $\Delta$ such that $D \cup \Delta$ is consistent, and the TSGD $(V, E, D \cup \Delta, L)$ is strongly-acyclic with respect to $R$?

**Proof:** See Appendix E. □

Note that, in an execution, at any instant, the invariant holds if and only if at that instant, in the TSGD $(V, E, D, L)$, there exists a set of dependencies $\Delta$ such that $(V, E, D \cup \Delta, L)$ is strongly acyclic with respect to $R$ (since every element in $R$ has arity 2, the TSGD is strongly acyclic with respect to $R$ if and only if no instantiations of regular terms in $R$ can result in $S$). Thus, from Theorem 8, it follows that determining if the invariant holds is NP-complete.

9 Conclusion

In an MDBS environment, based on the semantics of transactions, certain non-serializable executions are acceptable. In this paper, we proposed a simple and powerful mechanism for specifying, in an MDBS environment, the set of non-serializable executions that are unacceptable. The undesirable interleavings among global subtransactions are specified using regular expressions over the type of global subtransactions. We showed that using our mechanism, it is possible to characterize interleavings that cannot be captured by existing mechanisms for specifying interleavings. Also, unlike existing approaches, our approach scales well to the addition of new global applications in the system. We developed efficient graph-based schemes (optimistic and conservative) in order to ensure that the concurrent execution of transactions meet specifications. In MDBS environments in which certain non-serializable executions are acceptable, we expect our schemes to outperform existing schemes for ensuring global serializability. We showed that although none of the conservative schemes proposed by us permit optimal concurrency, the problem of optimally scheduling operations for execution is NP-complete. We are currently investigating recovery algorithms that can be used with our schemes to deal with failures and transaction aborts, alternative mechanisms for specifying interleavings, and distributed concurrency control schemes for preventing unacceptable interleavings. The results in this paper are also applicable to homogeneous distributed database systems.

References

**Definition 6:** Consider a TSGD containing the sequence of edges \((G_0, s_{i_0}) (s_{i_0}, G_1) \cdots (G_{n-1}, s_{i_{n-1}}) (s_{i_{n-1}}, G_0)\), \(n > 1\). This sequence of edges form a strong-cycle if

- for all \(j = 0, 1, \ldots, n - 1\), \(G_j \neq G_{(j+1) \mod n}\), \(s_{i_j} \neq s_{i_{(j+1) \mod n}}\) and dependency \((G_j, s_{i_j}, G_{(j+1) \mod n}, s_{i_{(j+1) \mod n}}) \notin D\).
- \(D \cup \{(G_{(j+1) \mod n}, s_{i_j}) \mapsto (s_{i_j}, G_j) : 0 \leq j \leq n - 1\}\) is consistent.

A TSGD is said to be strongly-acyclic if it does not contain any strong-cycles. \(\Box\)

Strong-minimality is defined in terms of strong-cycles as follows.

**Definition 7:** A set of dependencies \(\Delta\) is strongly-minimal with respect to the TSGD and transaction \(G_i \in V\) iff

- \((V, E, D \cup \Delta, L)\) does not contain any strong-cycles involving \(G_i\), and
- for all \(d \in \Delta\), \((V, E, D \cup \Delta - d, L)\) contains a strong-cycle involving \(G_i\). \(\Box\)

The computation of a minimal \(\Delta\) can be shown to be NP-hard as a consequence of the following NP-hardness result.

**Theorem 7:** Given a TSGD \((V, E, D, L)\) such that \(D\) is consistent, and a transaction \(G_i \in V\) such that for all transactions \(G_j \in V\), for all sites \(s_k\), dependency \((G_i, s_k) \mapsto (s_k, G_j) \notin D\). A TSGD \((V', E', D', L')\) resulting due to the deletion of \(G_i\), its edges and dependencies from \((V, E, D, L)\) is strongly-acyclic. The problem of computing a set of dependencies, \(\Delta\), such that \(D \cup \Delta\) is consistent and \(\Delta\) is strongly-minimal with respect to the TSGD and transaction \(G_i\) is NP-hard.

**Proof:** See Appendix E. \(\Box\)

If the regular specification were to contain the single regular term \(RT = (A : a, a) : (A : a, a)\) every global transaction were to have type \(A\), and one or more subtransactions of type \(a\), then a minimal \(\Delta\) would also be strongly-minimal with respect to the TSGD and \(G_i\) (since an instantiation of \(RT\) is a valid program). Any violation of \(\Delta\) involving \(G_i\) could result in a program crash if and only if there is a strong-cycle in the TSGD containing \(G_i\). Thus, from Theorem 7, it follows that the computation of a minimal \(\Delta\) is NP-hard.

Also, in the TSGS scheme presented in the previous section, the set of dependencies in order to prevent instantiations of regular terms, are computed when an \(init_i\) operation is processed. However, this approach that involves restricting the execution of \(ser_k(G_i)\) operations \textit{a priori} when \(init_i\) is processed is inflexible and could result in a low degree of concurrency. An alternative conservative scheme would be one that does not impose restrictions on the processing of \(ser_k(G_i)\) operations when \(init_i\) is processed, but instead, ensures that at any instant the following invariant holds: \(\forall S \in S\) there exists a total order on unexecuted \(ser_k(G_i)\) operations such that executing them in the order consistent with the total order does not result in instantiations of regular terms in \(S\). The invariant ensures that all the unexecuted \(ser_k(G_i)\) operations can be executed without jeopardizing the correction of \(S\) and without aborting any global transactions. Furthermore, a \(ser_k(G_i)\) operation is permitted to execute if and only if its execution preserves the invariant (note that the processing of \(init_i\) operation trivially preserve the invariant). Thus, the alternative approach would provide the maximum degree of concurrency that can be provided by a conservative scheme. However, it can be shown that, at any instant, determining if the invariant holds is an NP-complete problem. We begin by showing a related problem, NP-complete, for which we need the following definition.
for every pair of nodes \((w, u)\) such that \((w, v')\) and \((u, v')\) are both edges in the TSGD, \((w, u)\) being appended to \(\text{anc}(v')\) when \(v'\) is visited in state \(st\).

The dependencies added to the TSGD during the processing of an \(\text{init}_i\) operation ensure that instantiations of regular terms involving global transaction \(G_i\) are possible. Thus, it can be shown by a simple induction argument on the number of \(\text{init}_i\) operations processed that there are no instantiations of any of the regular terms in global schedule \(S\).

**Theorem 5:** Let \(R\) be a complete regular specification. The TSGD scheme ensures that \(R\) is correct with respect to \(R\).

**Proof:** See Appendix D. \(\Box\)

The complexity of the TSGD scheme is dominated by the number of steps it takes to process an \(\text{init}_i\) operation. \(\text{Detect}_{\text{Ins}}\text{TSGD1}\) can be shown to terminate in \(O(n_Sn_G^2m)\) steps and \(\text{Detect}_{\text{Ins}}\text{TSGD2}\) can be shown to terminate in \(O(n_Sn_G^2m)\) steps. Since, in the worst case, \(\text{Detect}_{\text{Ins}}\text{TSGD2}\) is involved for every regular term in \(R\) and for every subtransaction of \(G_i\), the complexity of the TSGD scheme is as stated in the following theorem.

**Theorem 6:** The worst-case complexity of the TSGD scheme is \(O(n_Sn_G^2mn_Rv_S)\) if \(\text{Detect}_{\text{Ins}}\text{TSGD1}\) is \(\text{Detect}_{\text{Ins}}\text{TSGD1}\) and is \(O(n_Sn_G^3mn_Rv_S)\) if \(\text{Detect}_{\text{Ins}}\text{TSGD2}\) is \(\text{Detect}_{\text{Ins}}\text{TSGD2}\). \(\Box\)

Among the conservative schemes presented in the last two sections, the TSGD scheme with \(\text{Detect}_{\text{Ins}}\text{TSGD2}\) provides the highest degree of concurrency, but also has the highest complexity. The TSGD scheme with \(\text{Detect}_{\text{Ins}}\text{TSGD1}\) has the lowest complexity among all the schemes, but also permits the lowest degree of concurrency. The TSGD scheme with \(\text{Detect}_{\text{Ins}}\text{TSGD1}\) and the TSG scheme with \(\text{Detect}_{\text{Ins}}\text{TSGD2}\) have identical complexities, but the degree of concurrency provided by the TSG schemes is incomparable.

### 8 Intractability Results

In the TSGD scheme in the previous section, the set of dependencies \(\Delta\) computed during the processing of an \(\text{init}_i\) operation ensures that there will be no instantiations of regular terms in \(S\) involving global transaction \(G_i\). However, a number of the restrictions imposed on the processing of \(\text{ser}_k(G_i)\) operations due to the dependencies in \(\Delta\) may be unnecessary; that is, there may exist a set of dependencies \(\Delta' \subseteq \Delta\) such that adding \(\Delta'\) to the TSGD prevents instantiations of regular terms in \(S\) involving \(G_i\).

Let us refer to a set of dependencies \(\Delta\) as \textit{minimal} if \(\Delta\) ensures that there will be no instantiations of regular terms in \(S\) involving \(G_i\), while for any \(\Delta' \subseteq \Delta\), adding \(\Delta'\) to the TSGD does not prevent such instantiations. Thus, ideally, in order to impose minimal restrictions on the execution of \(\text{ser}_k(G_i)\) operations, the set of dependencies \(\Delta\) computed when an \(\text{init}_i\) operation is processed must be minimal. However, the computation of such a minimal \(\Delta\) is NP-hard. In order to show this, we need to first define \textit{strong-minimality} for which we need the following additional definitions.

**Definition 5:** A set of dependencies \(D\) is consistent, if there do not exist nodes \(v_0, v_1, \ldots, v_n\), \(n > 2\), in the TSGD such that \((v_1, v_0)\rightarrow(v_0, v_2)\in D\), \((v_2, v_0)\rightarrow(v_0, v_3)\in D\), \ldots, \((v_{n-1}, v_0)\rightarrow(v_0, v_1)\in D\).

In addition, we need to define the notion of a \textit{strong-cycle}.
\( G_i \) and its edges are inserted into the TSGD. For every operation \( \text{ser}_k(G_i) \in G_i \), for all transactions \( G_j \in V \) such that \( \text{ser}_k(G_j) \in G_j \) and \( \text{act}(\text{ser}_k(G_j)) \) has executed, dependencies \((G_j, s_k)\rightarrow(s_k, G_j)\) are added to \( D \).

\[ \Delta := \emptyset; \]
for every regular term \( RT = e_0 : \text{reg.exp} \) in \( R \) such that \( \text{type}(G_i) = \text{hdr}(e_0) \)
for every subtransaction \( G_{i;k} \) such that \( \text{type}(G_{i;k}) = \text{last}(e_0) \)
begin
if \( \text{arity}(e_0) = 1 \) then \( \text{set}_1 := \{s_k\} \)
else \( \text{set}_1 := \{s_i : (s_i \neq s_k) \land \text{type}(G_{i;i}) = \text{first}(e_0)\} \);
\( \Delta := \Delta \cup \text{Detect.Ins.TSGD?((V, E, D \cup \Delta, L), G_i, s_k, \text{set}_1, RT)} \);
end;
\( D := D \cup \Delta \)

Figure 9: Pseudocode for \( \text{act}(\text{init}_i) \)

- \( \text{act}(\text{ser}_k(G_i)) \): For every transaction \( G_j \in V \) such that \( \text{ser}_k(G_j) \in G_j \) and \( \text{act}(\text{ser}_k(G_j)) \) not yet been executed, dependencies \((G_i, s_k)\rightarrow(s_k, G_j)\) are added to \( D \). Operation \( \text{ser}_k(G_i) \) submitted to the local DBMSs (through the servers) for execution.

- \( \text{cond}(\text{ack}(\text{ser}_k(G_i))) \): \( \text{true} \).

- \( \text{act}(\text{ack}(\text{ser}_k(G_i))) \): Operation \( \text{ack}(\text{ser}_k(G_i)) \) is sent to GTM1.

- \( \text{cond}(\text{fin}_i) \): \( \text{true} \).

- \( \text{act}(\text{fin}_i) \): For every transaction \( G_j \in V \) such that \( \text{val}_j \) has been processed, if for every transac-
tion \( G_k \in V \) serialized before \( G_j \), \( \text{val}_k \) has been processed, then \( G_j \) along with all its edges dependencies is deleted from the TSGD.

Procedures \text{Detect.Ins.TSGD1} \ and \text{Detect.Ins.TSGD2} \ traverse edges in the TSGD in order to
detect potential instantiations, and are very similar to procedures \text{Detect.Ins.TSG1} \ and \text{Detect.Ins.TSG2}, respectively. \text{Detect.Ins.TSGD1} \ and \text{Detect.Ins.TSGD2} \ are similar to \text{Detect.Ins.TSG1} \ and \text{Detect.Ins.TSG2}, in that they may detect false instantiations (\text{Detect.Ins.TSGD1} detects more false instantiations than \text{Detect.Ins.TSGD2}). However, instead of returning a set of site nodes, they return a set of dependencies that if added to the TSGD, restricts the execution of the appropriate \text{ser}_k \ operations so that there are no instantiations involving \( G_i \).

The updates to \( \text{anc}(v') \) \ and \( \text{V.set}(v') \) when a node \( v' \) is visited are the same in both \text{Detect.Ins.TSGD1} \ and \text{Detect.Ins.TSG1}. The main differences between the two schemes is that for the sequence of edges \((v_0, u_0), (v_1, u_1), \ldots, (v_p, u_p)\) traversed as mentioned earlier, \text{Detect.Ins.TSGD1} \ ensures that for all \( i = 1, 2, \ldots, p \), \((v_i, u_i)\) is distinct from \((v_0, u_0)\) (unlike \text{Detect.Ins.TSG1}, which only ensures that \((v_0, u_0)\) and \((v_p, u_p)\) are distinct). Furthermore, due to the presence of dependencies in the TSGD, \( v' \) due to conditions in steps 3(a) and 3(b), for any state \( st \) of \( F \), every node \( v' \) in the TSGD may need to be visited in state \( st \) once for each node \( w \) such that \( (v', w) \) is an edge in the TSGD, \( w \) being appended to \( \text{anc}(v') \) when \( v' \) is visited in state \( st \).

\text{Detect.Ins.TSGD2}, too, updates \( \text{anc}(v') \) \ and \( \text{V.set}(v') \), when a node \( v' \) is visited, in a manner identical to \text{Detect.Ins.TSG2} \, and like \text{Detect.Ins.TSG2}, ensures that for the sequence of edges \((v_0, u_0), (v_1, u_1), \ldots, (v_p, u_p)\), for all \( i = 1, 2, \ldots, p \), \((v_i, u_i)\) is distinct from both \((v_{i-1}, u_{i-1})\) and \((v_0, u_0)\). Also, due to the presence of dependencies in the TSGD, and due to conditions in steps 3(a), 3(b), 3(c), ...
in the same state $st$ multiple times, each time the same node $u$ being appended to $\text{anc}(v')$, $(st, u, v')$ is added to $V\_set(v')$ when $v'$ is visited in state $st$ and $u$ is appended to $\text{anc}(v')$. 

Similarly, in $\text{Detect\_Ins\_TSG2}$, since the TSG does not contain any dependencies, and due to the first two conditions in Step 3, in order to detect instantiations, for any state $st$ of $F$, every node $u$ in the TSG may need to be visited in state $st$ twice for each node $w$ such that $(v', w)$ is an edge in the TSG, the pairs $(w, u_1)$ and $(w, u_2)$ appended to $\text{anc}(v')$ the two times $v'$ is visited in state $st$ be distinct. Also, in order to prevent a node $v'$ from being visited in the same state $st$ multiple times, each time the same pair $(w, u)$ being appended to $\text{anc}(v')$, $(st, (w, u))$ is added to $V\_set(v')$ when $v'$ is visited in state $st$ and $(w, u)$ is appended to $\text{anc}(v')$.

When an $\text{init}_i$ operation is processed, in order to prevent instantiations involving transactions that are not in the TSG scheme restricts the execution of certain $\text{ser}_k(G_i)$ operations by marking them (processing marked operations is delayed until all the operations ahead of it in the queue have been processed). Thus, by a simple induction argument on the number of $\text{init}_i$ operations processed, it can be shown that there are no instantiations of any of the regular terms in global schedule $S$ involving any of the global transactions.

**Theorem 3:** Let $R$ be a complete regular specification. The TSG scheme ensures that $S$ is correct with respect to $R$.

**Proof:** See Appendix C. $\square$

The complexity of the TSG scheme is dominated by the number of steps it takes to process an $\text{init}_i$ operation. Procedures $\text{Detect\_Ins\_TSG1}$ and $\text{Detect\_Ins\_TSG2}$ can be shown to terminate in $O(n_S n_G)$ and $O(n_S n_G^2 m)$ steps respectively. Since, in the worst case, when $\text{init}_i$ is processed, $\text{Detect\_Ins\_TSG}$ is invoked for every regular term in $R$ and for every subtransaction of $G_i$, the complexity of the TSG scheme is as stated in the following theorem.

**Theorem 4:** The worst-case complexity of the TSG scheme is $O(n_S n_G m n_R V_S)$ if $\text{Detect\_Ins\_TSG}$ is $\text{Detect\_Ins\_TSG1}$ and is $O(n_S n_G^2 m n_R V_S)$ if $\text{Detect\_Ins\_TSG}$? is $\text{Detect\_Ins\_TSG2}$. $\square$

7 Conservative Schemes with Dependencies

The conservative schemes described in the previous section do not exploit the knowledge of the serialization order of global subtransactions since they utilize the TSG as a data structure. In this section, we present conservative schemes that employ the TSGD as a data structure to store the execution order of $\text{ser}_k(G_i)$ operations and thus, provide a higher degree of concurrency than the schemes described in the previous section. In the schemes, if potential instantiations of regular terms involving $G_i$ is detected when the edges of the TSGD are traversed during the processing of an $\text{init}_i$ operation, the appropriate dependencies are added to the TSGD in order to restrict the processing of certain $\text{ser}_k(G_i)$ operations. We now specify, for every operation $o_j$ in QUEUE, $\text{cond}(o_j)$ and $\text{act}(o_j)$ (no action performed when a $\text{val}_i$ operation is processed, and $\text{cond}(\text{val}_i) = \text{true}$). Initially, for the TSGD, $V = E = \emptyset, D = \emptyset$. 

- $\text{cond}(\text{init}_i)$: true.
- $\text{act}(\text{init}_i)$: The pseudocode in Figure 9 is executed. Procedure $\text{Detect\_Ins\_TSG}$ can be either $\text{Detect\_Ins\_TSG1}$ (see Figure 13 in Appendix A) or $\text{Detect\_Ins\_TSG2}$ (see Figure 13 in Appendix A).

For all transactions $G_i$ in $\text{Val}$, there exists $(G_i, o_j) = (G_i, \text{val}_i)$. But $\text{val}_i$
Given $G_i$ and its edges are inserted into the TSG. Also, for every operation $ser_k(G_i) \in G_i$, $ser_k(G_i)$ is inserted at the end of the queue for site $s_k$.

\[
\text{set}_2 := \emptyset; \\
\text{for every regular term } RT = e_0 : \text{reg}_\text{exp} \text{ in } R \text{ such that } type(G_i) = hdr(e_0) \text{ do } \\
\quad \text{for every subtransaction } G_{ik} \text{ such that } type(G_{ik}) = last(e_0) \text{ do } \\
\quad\quad \text{begin } \\
\quad\quad\quad \text{if } \text{arity}(e_0) = 1 \text{ then } \text{set}_1 := \{s_k\} \quad \text{else set}_1 := \{s_l : (s_l \neq s_k) \land (type(G_{il}) = first(e_0))\} \\
\quad\quad\quad \text{set}_2 := \text{set}_2 \cup \text{Detect}_\text{Ins}_TSG?((V, E, L), G_i, s_k, \text{set}_1, \text{set}_2, RT) \\
\quad \text{end } \\
\text{For every site } s_l \text{ in } \text{set}_2, \text{ser}_l(G_i) \text{ is marked in the queue for site } s_l.
\]

Figure 8: Pseudocode for act(init;)

- act(fin;): For every transaction $G_j \in V$ such that $val_j$ has been processed, if for every transaction $G_k \in V$ such that there is a path from $G_j$ to $G_k$ in the TSG, $val_k$ has been processed, then $G_j$, along with all its edges, is deleted from the TSG.

Procedures Detect_Ins_TSG1 and Detect_Ins_TSG2 traverse edges in the TSG in order to detect potential instantiations in a similar fashion as Detect_Ins_Opt. However, unlike Detect_Ins_Opt which returns commit/abort, they return a set of site nodes that identify $ser_k(G_i)$ operations whose execution, if restricted, would prevent instantiations of regular terms. Both Detect_Ins_TSG1 and Detect_Ins_TSG2 may detect false instantiations and thus require the execution of more operations to be restricted than are actually required to prevent instantiations (Detect_Ins_TSG1 detects a larger number of false instantiations than Detect_Ins_TSG2, but has a lower complexity than Detect_Ins_TSG2). Also, in Detect_Ins_TSG1 and Detect_Ins_TSG2, since the TSG contains no dependencies, when a new $v'$ is visited, the nodes appended to $anc(v')$ are different from those appended in Detect_Ins_Opt.

As mentioned earlier, for an instantiation $t_0 : t_1 \cdots t_{n-1}$, if for some $j, k, j = 0, 1, 2, \ldots, n > 0$, $j < k < j+n$, it is the case that for all $l, j < l < k, \text{arity}(t_{l\text{mod}n}) = 1$, then $\text{hdr}(t_j) \neq \text{hdr}(t_{(j+1)\text{mod}n})$. Thus, ideally, an algorithm for precisely detecting instantiations must ensure that if it does a 2-arity traversal of an edge $(v_0, u_0)$ followed by a sequence of 1-arity traversals of edges $(v_1, u_1), \ldots, (v_{p-1}, u_{p-1})$ and finally a 2-arity traversal of edge $(v_p, u_p)$, then all the edges traversed $(v_0, u_0), (v_1, u_1), \ldots, (v_p, u_p)$, are distinct. Detect_Ins_Opt ensures that the edges $(v_0, u_0), \ldots, (v_p, u_p)$, are distinct since there are dependencies between any two edges of the TSGD that is passed as argument to Detect_Ins_Opt. However, in case there are no dependencies between certain edges of the TSGD, the computational complexity of such an ideal algorithm that ensures $(v_0, u_0), \ldots, (v_p, u_p)$, are distinct would be prohibitive (the problem of precisely detecting instantiations by traversing edges in the TSG is intractable). Thus, procedure Detect_Ins_TSG1 only ensures that edges $(v_0, u_0), (v_1, u_1), \ldots, (v_p, u_p)$ are distinct, while procedure Detect_Ins_TSG2 goes one step further and ensures that for $i = 1, \ldots, p, (v_i, u_i)$ is distinct from both $(v_0, u_0)$ and $(v_{i-1}, u_{i-1})$. For this purpose, Detect_Ins_TSG1 appends to $anc(v')$, when $v'$ is visited, the node $u$ such that $(u, v')$ is the most recent 2-arity traversed edge by Detect_Ins_TSG1; while Detect_Ins_TSG2 appends to $anc(v')$, when $v'$ is visited, the ordered pair of nodes $(u, w)$ such that $(u, v')$ is the most recent 2-arity traversed edge and $(w, v')$ is the most recently traversed edge.

In Detect_Ins_TSG1, since the TSG does not contain any dependencies, and due to the condition in Step 3(a), in order to detect instantiations, for any state $st$ of $F$, every node $v'$ in the TSG may not be visited in state $st$ twice during the execution of Detect_Ins_TSG1; the nodes appended to $anc(v')$ in Detect_Ins_TSG2 in the TSGD.
Theorem 1: Let $R$ be a complete regular specification. The optimistic scheme ensures that $R$ correct with respect to $R$.

Proof: See Appendix B. □

The complexity of the optimistic scheme is dominated by the number of steps it takes to process a $val_i$ operation. Procedure Detect.Ins.Opt can be shown to terminate in $O(n_R n_G^2 m)$ steps. Since the worst case, Detect.Ins.Opt is invoked for every regular term in $R$ and for every subtransaction in $G_i$, the complexity of the optimistic scheme is as stated in the following theorem.

Theorem 2: The worst-case complexity of the optimistic scheme is $O(n_R n_G^2 m n_R v_S)$. □

Note that $n_R$, $n_S$ and $v_S$ can be expected to be small in comparison to the number of global transactions $n_G$ and the number of sites $m$ in the MDBS environment. Also, in our complexity analysis of the optimistic scheme, we assume that Detect.Ins.Opt can be implemented such that each of the three conditions in Step 2 can be checked in constant time.

6 Conservative Schemes

In this section, we present conservative schemes for ensuring that global schedule $S$ is correct. These schemes utilize a data structure referred to as the transaction-site graph (TSG). A TSG is similar to TSGD, except that it contains no dependencies. Thus, a TSG is a 3-tuple $(V, E, L)$. Also, associated with every site $s_k$, is a queue. Initially, all queues are empty, and for the TSG, both $V = \emptyset$ and $E = \emptyset$.

When $init_i$ is processed, edges in the TSG are traversed in order to detect potential instantiations of regular terms involving $G_i$. In case potential instantiations are detected, the processing of the $ser_k(G_i)$ operation in the queues is constrained by “marking” them. For an operation $o_j$ in QUEUE cond$(o_j)$ and act$(o_j)$ are defined as follows (no actions are performed when a $val_i$ operation is processed and cond$(val_i)$ is true):

- **cond**(init$_i$): true.

- **act**(init$_i$): The pseudocode in Figure 8 is executed. Procedure Detect.Ins.TSG? in the pseudocode can be either Detect.Ins.TSG1 (see Figure 11 in Appendix A) or Detect.Ins.TSG2 (see Figure 12 in Appendix A) (the two procedures differ in the degree of concurrency they permit and their complexities).

- **cond**(ser$_k(G_i)$): For every transaction $G_j \in V$ such that $ser_k(G_j) \in G_j$, if $act(ser_k(G_j))$ has executed, then $act(ack(ser_k(G_j)))$ has also completed execution. In addition, if $ser_k(G_i)$ is marked, then it is the first element in the queue for site $s_k$.

- **act**(ser$_k(G_i)$): Operation $ser_k(G_i)$ is submitted to the local DBMSs (through the servers) for execution.

- **cond**(ack(ser$_k(G_i)$)): true.

- **act**(ack(ser$_k(G_i)$)): Operation $ser_k(G_i)$ is deleted from the queue for site $s_k$ (note that $ser_k(G_i)$ may not be at the front of the queue for site $s_k$). Operation $ack(ser_k(G_i))$ is sent to GTM$_1$.

- **cond**(fin$_i$): true.
Let \((V', E', D', L')\) be the TSGD obtained as a result of deleting from \((V, E, D, L)\), the edges and dependencies incident on transactions \(G_k \in V, G_k \neq G_i\), such that \(val_k\) has not yet been processed.

for every regular term \(RT = c_0 : \text{reg.exp} \) in \(R\) such that \(\text{type}(G_i) = \text{hdr}(c_0)\) do

for every subtransaction \(G_{ik}\) such that \(\text{type}(G_{ik}) = \text{last}(c_0)\) do

\[
\begin{align*}
\text{begin} & \\
\text{if } \text{arity}(c_0) = 1 & \text{ then } set_1 := \{s_k\} \\
\text{ else } set_1 & := \{s_l : (s_l \neq s_k) \land (\text{type}(G_{il}) = \text{first}(c_0))\}; \\
\text{if Detect}_{\text{Ins}}_{\text{Opt}}((V', E', D', L'), G_i, s_k, set_1, RT) = \text{abort} & \text{ then} \\
\text{begin} & \\
\text{Delete } G_i \text{ along with all its edges and dependencies from the TSGD;} & \\
\text{Inform GTM}_1 \text{ to abort } G_i; & \\
\text{exit()} & \text{ end} \\
\text{end} & \\
\text{Inform GTM}_1 \text{ to commit } G_i; & \\
\end{align*}
\]

Figure 7: Pseudocode for \(\text{act}(\text{val}_i)\)

and element \(l_0, \text{head}(l)\) returns \(l_1, \text{tail}(l)\) returns \([l_2, \ldots, l_p]\) and \(l_0 \circ l\) returns \([l_0, l_1, l_2, \ldots, l_p]\). Also, an ordered pair \(o = (o_1, o_2), o[1] = o_1, \text{while } o[2] = o_2\).

Detect_{\text{Ins}}_{\text{Opt}} utilizes the finite automaton \(F = FA(RT)\) to ensure that the sequence of edges traversed by it corresponds to a string in \(L(\text{reg.exp})\). Every time Detect_{\text{Ins}}_{\text{Opt}} traverses an edge on the current state of \(F\) is updated and a node is visited (the node is said to be visited in a state equal to the current state of \(F\). A node may be visited multiple times during the execution of Detect_{\text{Ins}}_{\text{Opt}}. The current node being visited is stored in variable \(v\), while the current state of \(F\) is stored in variable \(\text{cur.st}\). Since instantiations may contain elements with arity 1, edge traversals do not always require changes in the TSGD.

Detect_{\text{Ins}}_{\text{Opt}} must be distinct. This is ensured by appending to the list \(\text{anc}(v')\), when \(v'\) is visited for the first time, any new node \(u\) such that \((v, v')\) is the most recently traversed edge. Furthermore, an edge is traversed only if it satisfies the condition in Step 3(a). Since the TSGD contains dependencies, and due to conditions in steps 3(a) and 3(b), in order to detect instantiations, for any state \(st\) of \(F\), every node \(v'\) in the TSGD is visited in state \(st\) at least once for every edge \((v', u)\) whose traversal could result in \(v'\) being visited in state \(st\). However, in order to prevent a node \(v'\) from being visited in the state \(st\) due to the traversal of the same edge \((v', u)\) multiple times, the ordered pair \((st, u)\) is added to \(V.set(v')\) when \(v'\) is visited in state \(st\) due to edge \((v', u)\) being traversed. Also, an edge must satisfy the condition in Step 3(c) before it can be traversed. Finally, every time a node \(v'\) is visited in the current node and the current state of \(F\) just before \(v'\) is visited is appended to \(F.list(v')\) to enable backtracking from \(v'\) to take place (Step 4 of procedure Detect_{\text{Ins}}_{\text{Opt}}).

When a \(val_i\) operation is processed, \(G_i\) is aborted if there is an instantiation of a regular term involving \(G_i\) and other transactions \(G_j\) that have already been committed in \(S\). Thus, by a simple induction, the TSGD can be reduced to a TSGD with only regular terms.
sites (site nodes) and global transactions in S (transaction nodes), E is a set of edges, D is a set of dependencies (denoted by \( \rightarrow \)) between edges incident on a common site node, and L is a function that maps every edge to an ordered pair \((\tau_1, \tau_2)\) where \(\tau_1 \in g\tau\) and \(\tau_2 \in l\tau\). Site and transaction nodes are labeled by the corresponding sites and transactions, respectively. Edges in the TSGD may only present actions between transaction nodes and site nodes. An edge between a transaction node \(G_i\) and a site node \(s_k\) is present only if operation \(ser_k(G_i) \in G_i\), and is denoted by either \((s_k, G_i)\) or \((G_i, s_k)\). The set of edges \(\{(G_i, s_k) : ser_k(G_i) \in G_i\}\) are referred to as \(G_i\)'s edges. Dependencies specify the relative order in which operations are processed and are sometimes also used to restrict the processing of \(ser_k(G_i)\) operations. If \((G_i, s_k)\) and \((s_k, G_j)\) are edges in the TSGD, then a dependency of the form \((G_i, s_k) \rightarrow (s_k, G_j)\) denotes that \(ser_k(G_i)\) is processed before \(ser_k(G_j)\). In addition, \(L\) maps the edge \((G_i, s_k)\) to the pair \((\text{type}(G_i), \text{type}(G_j))\).

The optimistic scheme uses the TSGD to store information relating to the serialization order of global subtransactions. Edges and dependencies are added to the TSGD when \(ser_k(G_i)\) operations are submitted for execution to the local DBMSs, and the TSGD is traversed in order to detect instantiations of regular terms when a \(val_i\) operation is processed. We now specify, for every operation \(o_j\) in \(\text{QUE}^i\), \(cond(o_j)\) and \(act(o_j)\) (no actions are performed when an \(init_i\) operation is processed, and \(cond(init_i)\) is \(true\)). Initially, for the TSGD, \(V = \emptyset\), \(E = \emptyset\), \(D = \emptyset\).

- \(cond(ser_k(G_i))\): For every transaction \(G_j\) such that \(ser_k(G_j) \in G_j\), if \(act(ser_k(G_j))\) has executed, then \(act(ack(ser_k(G_j)))\) has also completed execution.

- \(act(ser_k(G_i))\): Edge \((G_i, s_k)\) is inserted into the TSGD. For every transaction \(G_j \in V\) such that edge \((G_j, s_k) \in E\) and \(act(ser_k(G_i))\) has executed, dependency \((G_j, s_k) \rightarrow (s_k, G_i)\) is added to \(D\). For every transaction \(G_j \in V\) such that \((G_j, s_k) \in E\) and \(act(ser_k(G_j))\) has not yet been executed, dependency \((G_j, s_k) \rightarrow (s_k, G_j)\) is added to \(D\). Operation \(ser_k(G_i)\) is submitted to the local DBMSs (through the servers) for execution.

- \(cond(ack(ser_k(G_i)))\): \(true\).

- \(act(ack(ser_k(G_i)))\): Operation \(ack(ser_k(G_i))\) is sent to GTM1.

- \(cond(val_i)\): \(true\).

- \(act(val_i)\): The pseudocode in Figure 7 is executed. Procedure Detect_Insp in the pseudocode traverses the TSGD in order to detect instantiations of regular terms, and is as shown in Figure 7 in Appendix A.

- \(cond(fin_i)\): \(true\).

- \(act(fin_i)\): For every transaction \(G_j \in V\) such that \(val_j\) has been processed, if for every transaction \(G_k \in V\) serialized before \(G_j\), \(val_k\) has been processed, then \(G_j\), along with all its edges and dependencies, is deleted from the TSGD.

When a \(val_i\) operation is processed, for certain regular terms in \(R\), \(Detect_{Ins\_Opt}\) is invoked in order to determine if there is an instantiation of the regular term in \(R\) involving transaction \(G_i\) (i.e., that there is a dependency between every pair of edges in \((V', E', D', L'))\). Procedure Detect_Insp traverses edges in the TSGD, beginning with node \(G_i\) in a direction against that of dependencies in the TSGD in order to detect instantiations of \(RT\). If it detects an instantiation of \(RT\) involving transaction \(G_i\), \(Detect_{Ins\_Opt}\) returns abort. Since regular specifications are complete, any instantiation of a regular term \(RT\) involving transaction \(G_i\) can be detected by traversing the TSGD beginning at node \(G_i\). Exception: read list nodes in \(Detect_{Ins\_Opt}\) are defined for lists. For read lists, \(\text{Hash}\) is
Thus, \( \text{reg.exp}' \) is a regular expression over the alphabet \( \Sigma_F = \{(\tau_1, \tau_2) : (\tau_1 \in g\tau) \land (\tau_2 \in l\tau) \} \cup \{(\tau_1, \tau_2) : (\tau_1 \in g\tau) \land (\tau_2 \in l\tau) \} \). \( FA(RT) \) is a deterministic finite automaton that accepts exactly the strings in \( L(\text{reg.exp}') \). □

Note that transitions between states in \( FA(RT) \) are due to elements in \( \Sigma_F \). For a finite automaton \( F \), we denote the initial state by \( \text{init.st}_F \) and the state transition function by \( \text{st}_F \). We distinguish between elements with arity 1 and those with arity 2 in the construction of \( FA(RT) \) since the graph need to be traversed differently in the two cases. For the regular term \( RT = (\text{approx.listing} : \text{list}, \text{list}) : (\text{transfer} : \text{insert}, \text{delete})^+ \), the finite automaton \( FA(RT) \) is as shown in Figure 6 (note that \( s_2 \) is the accept state).

We define the complexity of a concurrency control scheme to be the average number of steps it takes the scheme to schedule all the operations associated with global transaction \( G_i \). For the purposes of analyzing the complexity of the various schemes, we assume the following.

- The average number of sites at which a global transaction executes is \( v_S \).
- At no point during the execution of a scheme does the difference between the number of \( i;_i \) and \( f;_i \) operations processed by the scheme exceed \( n_G \) (we assume that \( v_S \ll n_G \)).
- The number of regular terms in the regular specification \( R \) is \( n_R \).
- Let \( RT \) be the regular term in \( R \) such that \( FA(RT) \) has the maximum number of states. We denote by \( n_S \), the number of states in \( FA(RT) \).

In the following sections, we present an optimistic scheme based on the certifier approach, conservative schemes for ensuring that \( S \) is correct. The optimistic scheme we present provides a higher degree of concurrency than the conservative schemes but could result in global transactions aborts that could hurt performance. (A concurrency control scheme, say \( CC_1 \), is said to provide a higher degree of concurrency than another concurrency control scheme \( CC_2 \) if, for any given order of insertion of operations into QUEUE by GTM, \( CC_2 \) does not cause a fewer number of operations to be added to WAIT than \( CC_1 \).) We specify the concurrency control schemes by specifying the data structures maintained by the scheme, and \( cond(o_j), act(o_j) \) for the various operations. We also specify the complexity of each of the schemes, and compare the degree of concurrency provided by the various schemes. An analysis of the complexity of the schemes and proofs of their correctness can be found in the appendices.

### 5 An Optimistic Scheme

The optimistic scheme presented in this section utilizes a data structure, referred to as the \( \text{transaction site graph with dependencies} \) (TSGD), introduced in [MRB+92]. A TSGD is an undirected bi-partite graph.
procedure Basic_Scheme():
Initialize data structures;
while (true)
beg
Select operation $o_j$ from the front of QUEUE;
if $cond(o_j)$ then begin
  act($o_j$);
  while (there exists an operation $o_l \in$ WAIT such that $cond(o_l)$ is true)
    begin
      act($o_l$);
      WAIT := WAIT \{-o_l\}
    end
  else WAIT := WAIT \cup \{o_j\}
end

Figure 5: Basic Structure of Concurrency Control Schemes

The regular specification for the car rental example presented in Section 3 is not complete. However, the regular specification containing the following additional regular terms (in addition to those listed in Section 3) is complete. In the following regular terms, $R_1$ is the regular expression $((\text{accurate\_listing} : \text{list, list}) + (\text{approx\_listing} : \text{list, list}) + (\text{transfer} : \text{delete, insert}) + (\text{transfer} : \text{insert, delete, delete}) + (\text{booking} : \text{reserve, reserve}))$.

- $(\text{transfer} : \text{insert, delete}) : (\text{transfer} : \text{insert, delete})^* (\text{approx\_listing} : \text{list, list}) (\text{transfer} : \text{insert, delete})^*$
- $(\text{transfer} : \text{insert, delete}) : R_1^* ((\text{accurate\_listing} : \text{list, list}) + (\text{accurate\_listing} : \text{list})) R_1^*$
- $(\text{transfer} : \text{delete, insert}) : R_1^* ((\text{accurate\_listing} : \text{list, list}) + (\text{accurate\_listing} : \text{list})) R_1^*$
- $(\text{booking} : \text{reserve, reserve}) : R_1^* ((\text{accurate\_listing} : \text{list, list}) + (\text{accurate\_listing} : \text{list})) R_1^*$
- $(\text{approx\_listing} : \text{list, list}) : R_1^* ((\text{accurate\_listing} : \text{list, list}) + (\text{accurate\_listing} : \text{list})) R_1^*$

Given a regular specification, it is possible to automate the process of determining the regular terms that need to be added to it such that it becomes complete. Note that the addition of regular terms to a regular specification so that it is complete needs to be performed only once (when the system is configured) and can be handled off-line.

Also, in order to detect instantiations of a regular term $RT = e_0 : reg\_exp$, the algorithms traversing graphs need to determine if there exist global transactions and subtransactions such that the sequence of their types is a string in $L(reg\_exp)$. For this purpose, the schemes employ a finite automaton [HU79], denoted by $FA(RT)$, which is defined as follows.

**Definition 4:** Let $RT = e_0 : reg\_exp$ be a regular term. Let $reg\_exp'$ be the regular expression obtained from $reg\_exp$ by performing the following two steps (in the order mentioned).

1. Replace each occurrence of $(\tau_1 : \tau_2)$ in $reg\_exp$ by $(\tau_1, \tau_2)$.
Definition 3: A regular specification $R$ is said to be complete if for every regular term $RT_0 = \epsilon_0 : \text{reg}\cdot\text{exp}_1$ in $R$, the following is true: let $\epsilon_1 \epsilon_2 \cdots \epsilon_{n-1}$, $n > 1$, be any string in $L(\text{reg}\cdot\text{exp}_1)$ so that for all $i = 1, 2, \ldots, n - 1$, $\epsilon_i \in \Sigma$. For every $i$, $i = 1, 2, \ldots, n - 1$, there exists a regular term $RT_2 = \epsilon_i : \text{reg}\cdot\text{exp}_2$ in $R$, such that the string $\epsilon_{(i+1)\mod n} \cdots \epsilon_{(i+n-1)\mod n}$ is an element of $L(\text{reg}\cdot\text{exp}_2)$.
utility is discussed below). We now briefly describe the operations in QUEUE for an arbitrary global transaction $G_i$ and site $s_k$.

- **init**$_i$ : This operation is inserted into QUEUE by GTM$_1$ before any other operation belonging to $G_i$ is inserted into QUEUE.

- **ser$_k$($G_i$)** : This operation is inserted into QUEUE by GTM$_1$ in order to request the execution of operation ser$_k$($G_i$).

- **ack(ser$_k$($G_i$))** : This operation is inserted into QUEUE by the servers when the local DBMS has completed executing operation ser$_k$($G_i$).

- **val$_i$** : This operation is inserted into QUEUE by GTM$_1$ before the global transaction $G_i$ is committed (global subtransactions may have been committed, however) and after ack(ser$_k$($G_i$)) for all ser$_k$($G_i$) operations belonging to $G_i$ have been received by GTM$_1$.

- **fin$_i$** : This operation is inserted into QUEUE by GTM$_1$ after val$_i$ is inserted into QUEUE.

Note that the init$_i$, val$_i$ and fin$_i$ operations do not belong to global transaction $G_i$.

Figure 4 illustrates the basic structure of GTM$_2$. CC is any concurrency control scheme for ensuring the correctness of $S$. CC selects operations from the front of QUEUE, in order to process them. Associated with CC are certain data structures (DS) that are manipulated every time an operation selected from QUEUE is processed by it. In addition, the following actions are performed by CC while it processes an operation $o_j$ in QUEUE.

- **init**$_i$ : Operation init$_i$ contains information relating to global transaction $G_i$ (e.g., the type of $G_i$ and its subtransactions, the set of sites, if known a priori, at which $G_i$ executes). This information is added to DS and is utilized by CC to detect instantiations of regular terms in $G_i$.

- **ser$_k$($G_i$)** : Operation ser$_k$($G_i$) is submitted to the local DBMS of operation (through ...

---

**Figure 3: The GTM Components**
3.3 Serialization Functions

In order to develop concurrency control schemes for ensuring correctness, we utilize the notion of serialization functions introduced in [MRB+92], which is similar to the notion of serialization even [ED90]. Let $\sigma_k$ be the set of all global subtransactions in schedule $S_k$. A serialization function for $s_k$, is a function that maps every subtransaction in $\sigma_k$ to one of its operations such that for any pair of subtransactions $G_{ik}, G_{jk} \in \sigma_k$, if $G_i$ is serialized before $G_j$ in $S_k$, then $ser(G_{ik}) <_{S_k} ser(G_{jk})$.

For example, if the timestamp ordering (TO) concurrency control protocol is used at site $s_k$, the local DBMS at site $s_k$ assigns timestamps to transactions when they begin execution, then a serialization function that maps every subtransaction $G_{ik} \in \sigma_k$ to $G_{ik}$’s begin operation is a serialization function for $s_k$. For a site $s_k$, there may be multiple serialization functions. For example, if the local DBMS at $s_k$ follows the two-phase locking (2PL) protocol, then a possible serialization function for $s_k$ maps every subtransaction $G_{ik} \in \sigma_k$ to the operation that results in $G_{ik}$ obtaining its last lock. Alternatively, a function that maps every subtransaction $G_{ik} \in \sigma_k$ to the operation that results in $G_{ik}$ releasing its last lock is also a serialization function for $s_k$.

We denote by $ser_k$, any one of the possible serialization functions for site $s_k$.

By controlling the execution of $ser_k(G_i)$ operations, the GTM can control the serialization of global transactions at the various, and the correctness of global schedule $S$ can be ensured. Then, we split the GTM into two components, GTM$_1$ and GTM$_2$ (see Figure 3). GTM$_1$ utilizes the information on serialization functions for various sites in order to determine for every global transaction $G_i$, operations $ser_k(G_i)$, and submits them to GTM$_2$ for processing. The remaining global transaction operations (that are not $ser_k(G_i)$) are directly submitted to the local DBMSs (through the servers).

GTM$_2$ is responsible for ensuring that the operations submitted to it by GTM$_1$ execute at the local DBMSs in such a manner that $S$ contains no instantiations of regular terms. Our concern, however, the remainder of the paper, shall be the development of concurrency control schemes for GTM$_2$ to ensure $S$ is correct. Our schemes require only GTM$_2$ to be centrally located; GTM$_1$ can be distributed across the various sites in order to reduce the overhead involved in global transaction processing.

4 Structure of Concurrency Control Schemes

In this section, we describe the structure and complexity of concurrency control schemes employed by GTM$_2$ for ensuring the correctness of $S$ (the concurrency control model we adopt is similar to that presented in [MRB+92]). As mentioned earlier, GTM$_1$ submits the $ser_k(G_i)$ operations belonging to each global transaction $G_i$ to GTM$_2$. GTM$_1$ inserts these operations into a queue, QUEUE. In addition, for every global transaction $G_i$, GTM$_1$ inserts into QUEUE, the operations $initi, valu; and fini$ (which

$^2$Actually, any function that maps every subtransaction $G_{ik} \in \sigma_k$ to one of its operations that executes between $G_i$ obtaining its last lock at site $s_k$ and its release of its last lock is a valid serialization function for $s_k$.}
In the execution in Figure 1(a),

\[(G_0 : G_{00}, G_{0(n-1)}) \cdot (G_{n-1} : G_{(n-1)(n-1)}, G_{(n-1)(n-2)}) \cdots (G_1 : G_{11}, G_{10})\]

is an instantiation of the regular term

\[(\text{approx.listing} : \text{list, list}) : (\text{transfer} : \text{insert, delete})+\]

since \(\text{type}(G_0 : G_{00}, G_{0(n-1)}) = (\text{approx.listing} : \text{list, list})\), and for \(n > 1\),

\[\text{type}(G_{n-1} : G_{(n-1)(n-1)}, G_{(n-1)(n-2)}) \cdots \text{type}(G_1 : G_{11}, G_{10}) = (\text{transfer} : \text{insert, delete}) \cdots (\text{transfer} : \text{insert, delete})\]

which is a string in \(L((\text{transfer} : \text{insert, delete})+)\).

Note that, in an instantiation, there can be multiple occurrences of a global transaction. However, since local schedules are serializable and global transactions have a single subtransaction at each site in the instantiation \(t_0 : t_1 \cdots t_{n-1}\), any two adjacent global transactions \(hdr(t_j)\) and \(hdr(t_{(j+1)\mod n})\), \(j = 0, 1, \ldots, n-1\), are distinct. Actually, instantiations possess the following stronger property.

**Property 1:** In an instantiation \(t_0 : t_1, \ldots, t_{n-1}\), if for some \(j, k, j = 0, 1, 2, \ldots, n-1, j < k < j + k \mod n\), it is the case that for all \(l, j < l < k\), \(\text{arity}(t_{l\mod n}) = 1\), then \(hdr(t_j) \neq hdr(t_{(j+1)\mod n}) \neq \cdots \neq hdr(t_{k\mod n})\). Also, for all \(r, s, j \leq r < s \leq k\), \(last(t_{r\mod n})\) is serialized after \(first(t_{s\mod n})\) at the same site. \(\square\)

The set of all cycles in the serialization orders of global transactions characterized by a regular term in \(S\) is the set of all instantiations of the regular term in \(S\). Since we use regular terms to specify unacceptable interleavings among global subtransactions, none of the cycles characterized by regular terms in \(S\) must be contained in \(S\) if it is to be correct.

**Definition 2:** Let \(R\) be a regular specification. Global schedule \(S\) is correct with respect to \(R\) if for every regular term \(RT\) in \(R\), there are no instantiations of \(RT\) in \(S\). \(\square\)

Since regular specifications are based on regular expressions, they are a powerful, yet simple tool for capturing the set of unacceptable interleavings. We expect that in most MDBS applications, regular specifications will be adequate to specify the set of non-serializable executions that are unacceptable. The regular specification for the car rental example in Section 3 contains the following three regular terms.

1. \((\text{approx.listing} : \text{list, list}) : (\text{transfer} : \text{insert, delete})+\)

2. \((\text{accurate.listing} : \text{list, list}) : ((\text{accurate.listing} : \text{list, list}) + (\text{approx.listing} : \text{list, list}) + (\text{transfer} : \text{delete, insert}) + (\text{transfer} : \text{insert, delete}) + (\text{booking} : \text{reserve, reserve}))+)\)

3. \((\text{accurate.listing} : \text{list}) : ((\text{accurate.listing} : \text{list, list}) + (\text{approx.listing} : \text{list, list}) + (\text{transfer} : \text{delete, insert}) + (\text{transfer} : \text{insert, delete}) + (\text{booking} : \text{reserve, reserve}))+)\)

Term 1 characterizes the set containing only the unacceptable interleavings involving \text{approx.listing} and \text{transfer} transactions (the unacceptable interleaving in Figure 1(a) is in the set of interleavings characterized by regular term 1, while the acceptable interleaving in Figure 1(b) is not). Note that it is not possible to characterize the set containing only the unacceptable interleavings involving \text{approx.listing} and \text{transfer} transactions using any of the mechanisms for specifying interleavings proposed in [GM83, FO89]. The terms 2 and 3 characterize the set of unacceptable interleavings that involve...
characterizes a set of cycles in the serialization order of global transactions. Every string in \( L \) specifies the types and serialization orders of global transactions in cycles belonging to the set. For example, a cycle in the serialization orders of global transactions \( G_0, \ldots, G_{n-1} \) such that for all \( k = 0, \ldots, n \), \( G_k \) is serialized before \( G_{(k+1) \mod n} \) is in the set of cycles characterized by \( R \) if there exists a string \( \tau_0 \cdots \tau_{n-1} \) in \( L(R) \) (each \( \tau_i \) is a global transaction type) such that for all \( k = 0, \ldots, n \), \( \text{type}(G_k) = \tau_k \). Global schedule \( S \) is correct if it does not contain any of the cycles in the serialization order of global transactions characterized by regular terms in the regular specification.

We now formally define the syntax and semantics of regular specifications, and the correctness of global schedules. Let \( \Sigma \) denote the set

\[
\{(\tau_1 : \tau_2, \tau_3) : (\tau_1 \in g\tau) \land (\tau_2, \tau_3 \in l\tau)\} \cup \{(\tau_1 : \tau_2) : (\tau_1 \in g\tau) \land (\tau_2 \in l\tau)\}. 
\]

In the car rental example, \( (\text{approx}\_\text{listing} : \text{list}, \text{list}) \) is an element of \( \Sigma \); however, \( (\text{approx}\_\text{listing} : \text{transfer}) \) does not belong to \( \Sigma \) since \( \text{transfer} \notin l\tau \). A regular specification \( R \) is a finite set of terms referred to as regular terms, each having the following syntax:

\[ e_0 : \text{reg}\_exp \]

where \( e_0 \) is an element of \( \Sigma \) and \( \text{reg}\_exp \) is a regular expression over the alphabet \( \Sigma \).

In the car rental example, \( (\text{approx}\_\text{listing} : \text{list}, \text{list}) : (\text{transfer} : \text{insert}, \text{delete}) \) is a regular term. Note that a regular term contains only global transaction and global subtransaction types.

We do not include local transaction information in regular terms is that the GTM has no control of the execution order of local transactions since they execute outside the control of the GTM.

Each regular term characterizes a set of cycles in the serialization orders of global transactions (and thus, a set of interleavings among global subtransactions). Every cycle in the set contains a global transaction and global subtransactions with types mentioned in \( e_0 \), and strings in \( L(\text{reg}\_exp) \) capture the types and serialization orders of the remaining global transactions and subtransactions in the cycles.

In order to describe the set of cycles characterized by a regular term, we need to introduce the following additional notation. For the global schedule \( S \), \( \Sigma_S \) denotes the set

\[
\{(G_i : G_{ij}, G_{ik}) : G_{ij} \text{ and } G_{ik} \text{ are distinct subtransactions of a committed transaction } G_i \text{ in } S \} \cup \{(G_i : G_{ij}) : G_{ij} \text{ is a subtransaction of a committed transaction } G_i \text{ in } S \}.
\]

For an element \( e \in \Sigma_S \), if \( e = (e_1 : e_2, e_3) \), then \( \text{type}(e) = (\text{type}(e_1) : \text{type}(e_2), \text{type}(e_3)) \), while if \( e = (e_1 : e_2) \), then \( \text{type}(e) = (\text{type}(e_1) : \text{type}(e_2)) \). Also, for an element \( e \in \Sigma \cup \Sigma_S \), if \( e = (e_1 : e_2) \), then \( \text{hdr}(e) = e_1 \), \( \text{firs}(e) = e_2 \), \( \text{last}(e) = e_3 \) and \( \text{arity}(e) = 3 \), while if \( e = (e_1 : e_2) \), then \( \text{hdr}(e) = \text{firs}(e) \) and \( \text{last}(e) \) are both \( e_2 \), and \( \text{arity}(e) \) is 1.

In the following definition, we introduce the notion of an instantiation of a regular term in order to identify the set of cycles in the serialization orders of global transactions characterized by a regular term in the global schedule \( S \).

**Definition 1:** Let \( RT = e_0 : \text{reg}\_exp \) be a regular term and let \( S \) be a schedule. \( t_0 : t_1 t_2 \cdots t_n \), \( n > 1 \), is an instantiation of \( RT \) in \( S \) if

1. \( t_j \in \Sigma_S \), and
2. \( \text{last}(t_j) \) and \( \text{firs}(t_{(j+1) \mod n}) \) execute at the same site, and \( \text{last}(t_j) \) is serialized after \( \text{firs}(t_{(j+1) \mod n}) \) at the site, and

\( t_j \) and \( t_{(j+1) \mod n} \) are in string in \( L(\text{reg}\_exp) \).

\( e_0 : \text{reg}\_exp \) is the regular expression over \( \Sigma \).
a totally ordered set of read, write, begin and commit operations. The local schedule at a site, denoted by $S_k$, is the set of all operations (belonging to local transactions and global subtransactions that execute at $s_k$ with a total order $\prec_S$ on them. The global schedule $S$ is the set of all the local schedules $S_k$.

We assume that the GTM is centrally located, and controls the execution of global transactions (our schemes require only one operation belonging to every global subtransaction to be submitted to the centrally located GTM; the remaining global subtransaction operations can be submitted directly to the local DBMSs at the sites outside the control of the centrally located GTM. As a result, a number of problems with having a centrally located GTM are alleviated since most of the global transaction processing can be done locally at the sites). It communicates with the various local DBMSs by means of server processes (one per transaction per site) that execute at each site on top of the local DBMS (see Figure 2). We assume that the interface between the servers and the local DBMSs provides operations to be submitted by the servers to the local DBMSs, and the local DBMSs to acknowledge completion of operations to the servers. The local DBMSs do not distinguish between local transactions and global subtransactions executing at its site. In addition, each of the local DBMSs ensures that local schedules are serializable.

### 3.2 Regular Specifications

Before we develop schemes that permit certain acceptable interleavings in an MDBS environment, we need to develop a mechanism for specifying the set of unacceptable interleavings among transactions. Since local schedules are serializable and local transactions execute at a single site, any unacceptable interleaving among transactions must be due to an unacceptable cycle in the serialization order of global transactions. The set of unacceptable cycles in the serialization orders of global transactions can be characterized using regular specifications which are defined using regular expressions (we assume the reader is familiar with the syntax and semantics of regular expressions as described in [HU79]). For a regular expression $R$, $L(R)$ is the set of all strings denoted by $R$.

We begin by giving the intuition that underlies our definition of regular specifications, followed by which, we present a formal treatment. A regular specification consists of regular terms, each of which
containing an accurate listing transaction \( G_0 \) and any other transaction \( G_1 \) such that \( G_1 \) is serializable both before and after \( G_0 \) is unacceptable.

Note that an accurate listing transaction returns a precise listing of car records, but cannot interleave with other transactions at all; on the other hand, although an approx listing transaction returns an approximate listing of car records, its subtransactions can interleave freely with subtransactions of booking transactions and most interleavings involving approx listing and transfer transactions are acceptable. Thus, based on the semantics of transactions, certain interleavings are acceptable. In many cases, since accurate listings may not be required, approx listing transactions can be employed instead of accurate listing transactions, thus enhancing the degree of concurrency. In the next section, we develop a mechanism for specifying the set of undesirable interleavings between global subtransactions.

3 Correctness of Schedules

In this section, we define the correctness of schedules in terms of undesirable interleavings among global transactions. However, before defining the correctness of schedules, we first describe the MDBS model that we adopt.

3.1 The MDBS Model

An MDBS is a collection of pre-existing and autonomous local DBMSs located at sites \( s_0, s_1, \ldots, s_n \). Each local DBMS exports an interface consisting of procedures that can be invoked by the global transactions in order to access and manipulate data at the local DBMSs. We denote by \( \ell \tau \), the set of types of all the procedures exported by local DBMSs. Every global transaction \( G_i \) has a type denoted by \( \text{type}(G_i) \) and consists of one or more subtransactions, at most one per site, each resulting from invocation of a single local DBMS procedure. The set of global transaction types is denoted by \( \text{type} \) and the subtransaction of \( G_i \) at site \( s_k \) is denoted by \( G_{ik} \). The type of subtransaction \( G_{ik} \) is denoted by \( \text{type}(G_{ik}) \) and is the same as the type of the local DBMS procedure whose execution results in \( G_{ik} \). In the car rental example in the previous section, \( \ell \tau = \{ \text{reserve, delete, insert, list} \} \) and \( G_{ik} \) is the type of subtransaction.
• **list.** This procedure lists the information contained in records for a subset of cars in the database.

  Arguments to the `list` procedure include a predicate that identifies the subset of cars whose records are to be listed (e.g., all the cars classified as compact, or all the cars not reserved on a particular date).

  Since a single car cannot be associated with more than one branch (site) at any given time, the following simple global integrity constraint is introduced due to the integration of the local DBMSs:

  For every car, at most one local DBMS contains the record for the car.

  The above constraint ensures that a single car is not rented out to multiple clients on the same date. The integration facilitates the execution of global transactions that reserve cars at multiple local DBMSs, list car records at multiple local DBMSs, and transfer car records from one local DBMS to another (in case the number of available cars at a site fall below a threshold value, cars are transferred to the site from nearby sites). Global transactions invoke the exported local DBMS procedures mentioned above (every invocation results in a global subtransaction), and are one of the following:

  • **booking.** A transaction that reserves cars at multiple local DBMSs and consists of one or more `reserve` subtransactions.

  • **transfer.** A transaction that transfers car records from one local DBMS to another, and consists of a `delete` subtransaction and an `insert` subtransaction.

  • **approx_listing.** A transaction that is used to provide an approximate, conservative listing of car records at multiple local DBMSs and consists of one or more `list` subtransactions. Every `approx_listing` transaction is only required to see a consistent database state, that is, a state which satisfies the global integrity constraint. As a result, it is unacceptable for an `approx_listing` transaction to list the same car record as contained in two or more distinct local DBMSs. However, it is acceptable for an `approx_listing` transaction to not list certain car records contained in some local DBMSs.

  • **accurate_listing.** A transaction that is used to return a global snapshot of the state of the MDBC, and consists of one or more `list` subtransactions. Thus, every `accurate_listing` transaction requires that every other transaction in the MDBC be either serialized before it or after it, or (it is unacceptable for a transaction to be serialized both before and after an `accurate_listing` transaction).

  Since each local DBMS ensures serializability, global subtransactions execute in isolation. Therefore, if every local transaction in the MDBC environment preserves the global integrity constraint, then every non-serializable execution consisting of only local transactions, `booking` and `transfer` transactions, preserves the global integrity constraint, and is acceptable. However, certain non-serializable executions containing `approx_listing` or `accurate_listing` transactions are unacceptable.

  Consider an execution involving a global transaction $G_0$ of type `approx_listing` that lists all car records at sites $s_0$ and $s_{n-1}$, and a set of global transactions $G_1, \ldots, G_{n-1}$ of type `transfer`, where each $G_i$ transfers, among other records, a particular car record $r_0$ from site $s_{i-1}$ to $s_i$. Suppose that for all $i = 0, 1, \ldots, n - 1$, $G_i$ is serialized before $G_{i+1 \mod n}$ at site $s_i$. This execution sequence is depicted in Figure 1(a), where the arrows represent the serialized before relationship among global subtransactions, and the text following “:” are global subtransaction types. In the above execution, the `approx_listing` transaction $G_0$ lists the car record $r_0$ as contained in databases at both sites $s_0$ and $s_{n-1}$, and thus, sees an inconsistent database state. As a result, the execution in Figure 1(b) is in which for all layers
on regular expressions over transaction types, and many of the interleavings that can be specified using our approach cannot be specified using the approaches in [GM83, FO89]. The proposed mechanism assumes that the global transactions access data at local DBMSs via well-defined interfaces (in order to ensure that global transactions access their databases in a controlled and restricted fashion, local DBMSs export a fixed set of procedures that can be invoked by global transactions). The mechanism is flexible, and can even be used to specify that global schedules resulting from the concurrent execution of transactions must be serializable. Unlike existing mechanisms, it scales well to the addition of new global applications and local DBMS procedures in the system. Furthermore, the mechanism facilitates the development of efficient graph-based schemes (optimistic and conservative) for ensuring that the concurrent execution of transactions in an MDBS environment meet the specifications. MDBS environments in which certain non-serializable executions are permissible, our schemes provide a higher degree of concurrency than schemes to ensure global serializability. We examine the trade-off between the complexities and the degree of concurrency permitted by the various schemes. We also show that although none of the conservative schemes proposed by us are optimal, the problem of scheduling the operations of the various concurrently executing transactions in order to permit optimistic concurrency is NP-complete. The results in this paper are also applicable to homogeneous distributed database systems. Recovery algorithms in case of site failures, transaction aborts, etc. are outside the scope of this paper and are not discussed.

The remainder of the paper is organized as follows. In Section 2, we present an MDBS application in which the semantics of transactions can be exploited in order to relax the serializability requirement. In Section 3, we present our mechanism for specifying unacceptable interleavings in an MDBS environment and define correctness of global schedules. The concurrency control model we adopt is discussed in Section 4. In Section 5, we develop an optimistic scheme that ensures global schedules are correct. Conservative schemes presented in sections 6 and 7 prevent unacceptable interleavings among global subtransactions. In Section 8, we show that the problem of optimally scheduling operationsexecuted is NP-complete. Concluding remarks are offered in Section 9.

2 A Motivating Example

Consider a car rental company with branches at \( n \) geographically distributed sites \( s_0, s_1, \ldots, s_{n-1} \). Each branch has a database that contains information related to cars at the branch (one record per car). The information stored in a record for a car comprises of a unique car identifier for the car, the model and make of the car, its classification (compact, luxury, etc.), the dates the car has been reserved, the name and credit card number of the client that made the reservations etc.

Consider an MDBS environment in which the databases belonging to the various branches are integrated. We assume that every local DBMS ensures the serializability of schedules at its site, but each local DBMS exports the following procedures:

- **reserve.** This procedure reserves a car. Arguments to the reserve procedure include the classification of the car to be reserved, the name and credit card number of the person reserving the car, and the dates the car is to be reserved. The reserve procedure only utilizes information stored in the car record in the local DBMS in order to ensure that the car is not reserved by multiple clients on the same date.

- **delete.** This procedure deletes records corresponding to one or more cars from the database. Arguments to the delete procedure include a list of car identifiers of the cars whose records are to be deleted.

- **update.** This procedure updates records corresponding to cars currently in the database and
1 Introduction

A multidatabase system (MDBS) consists of a number of pre-existing and autonomous local database management systems (DBMSs) that are integrated to enable users of a local DBMS to access data residing at remote systems. The local DBMSs may follow different concurrency control protocols. The integration is transparent and must be accomplished without any modifications to the local DBMS software and to the pre-existing local applications. This is achieved by building on top of the local DBMSs, a software module, referred to as the global transaction manager (GTM), that is responsible for co-ordinating the execution of global transactions – transactions whose execution span multiple local DBMSs. Those transactions that execute at a single local DBMS are referred to as local transactions.

Since the GTM is built on top of local DBMSs, and no modifications are made to local DBMS software and to pre-existing local applications, local transactions execute outside the control of the GTM. As a result, the GTM has no knowledge of the indirect conflicts between global transactions due to the execution of local transactions, and the only way the GTM can enforce a particular ordering of global transactions at a local DBMS is by controlling the execution of certain operations belonging to a local transaction. This is achieved through a technique known as serializability\(^1\) as the notion of correctness, as is done in [Pu88, BS88, ED90, BST90, BGRS91, PRS91, GR91, MRB+ 92, Raz92], could result in a low degree of concurrency and adversely affect the performance of the system. Furthermore, in most MDBS environments, due to the autonomous nature of local DBMSs, integrity constraints between data items belonging to different local DBMSs may not be expected to be weak and few in number. As a result, serializability may not be required in order to preserve database consistency, and it would therefore be advantageous to come up with a weaker notion of correctness for such systems.

For MDBS environments, two correctness criteria that are weaker than serializability have been proposed – quasi-serializability (QSR) [DE89] and two-level serializability (2LSR) [MRKS91]. Both criteria require each local DBMS to generate serializable schedules. In addition, 2LSR requires the restriction of global schedules to only global transactions to be serializable, while QSR imposes the stronger condition that global schedules be equivalent to a schedule in which all global transactions execute serially. QSR and 2LSR schedules preserve database consistency if transactions and integrity constraints are of a restricted nature.

The degree of concurrency permitted by the correctness criteria proposed in [DE89, MRKS91] is limited since they restrict themselves to only read and write operations. In [GM83, GMS87, FO89], the authors adopt a different approach in which a transaction is assumed to consist of steps, each of which could be semantically richer than read and write operations. In [GM83, FO89], types are associated with transactions, and mechanisms that use the type information for specifying acceptable interleavings between steps are developed. The authors also develop protocols to ensure that only the specified interleavings are permitted. In [GM83], transaction types are grouped into compatibility sets which are used to specify interleavings. All the steps of transactions whose types belong to a single compatibility set can interleave freely, while steps of transactions whose types belong to different compatibility sets cannot interleave at all. In [FO89], acceptable interleavings are specified by specifying the set of types of transactions that can interleave between any two consecutive steps of a transaction. A schedule is correct if it is equivalent to a schedule in which steps are executed serially, and between any two consecutive steps of a transaction, only steps of those transactions that are permitted to interleave appear in the schedule. In [PL91, KB91, WA92] the authors propose schemes that exploit the semantics of applications in order to bound the inconsistency due to non-serializable executions.

The approach we adopt in this paper enables the semantics of transactions to be exploited, and is similar to the one used in [GM83, FO89]. However, our mechanism for specifying interleavings is based on the concept of a conflict serializability (CSR) [Pa86], which enables factors in the specified.
Exploiting Transaction Semantics in Multidatabase Systems

Rajeev Rastogi\textsuperscript{1*}
Henry F. Korth\textsuperscript{2}
Avi Silberschatz\textsuperscript{1*}

\textsuperscript{1}Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

\textsuperscript{2}Matsushita Information Technology Laboratory
182 Nassau Street, third floor
Princeton, NJ 08542-7072

Abstract

\textit{Serializability} is the traditionally accepted notion of correctness in most database systems. However, in a \textit{multidatabase system} (MDBS) environment consisting of pre-existing and autonomous database systems, requiring schedules to be serializable could severely hurt performance. Besides, in a number of instances, the semantics of transactions can be exploited in order to permit certain non-serializable executions. In this paper, we propose a powerful, yet simple mechanism for specifying, in an MDBS environment, the set of non-serializable executions that are unacceptable. The undesirable interleavings among transactions are specified using \textit{regular expressions} over transaction types. The mechanism facilitates the development of efficient graph-based schemes for ensuring that the concurrent execution of transactions in the MDBS environment meet the specifications. We examine the trade-off between the complexities and the degree of concurrency permitted by the various schemes. Finally, we show that the problem of scheduling operations for execution so as to permit optimal concurrency is NP-complete.
EXPLOITING TRANSACTION SEMANTICS
IN MULTIDATABASE SYSTEMS

Rajeev Rastogi
Henry F. Korth
Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-45           December 1992