


Definition 8: Let $RT = e_0 : reg.exp$ be a regular term containing only elements with arity 2 and 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

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

A TSGD is said to be strongly-acyclic with respect to a regular specification $R$ iff for every $RT \in R$, 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 $D$ 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 types 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 site 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 $(G_0, s_0), (s_0, G_1), \ldots, (G_{n-1}, s_{n-1})$, $(s_{n-1}, G_0)$, $n > 1$. This sequence of edges forms 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 $(G_j, s_{i_j}) \rightarrow (s_{i_{j}}, G_{(j+1) \mod n}) \notin D$.

- $D \cup \{(G_{(j+1) \mod n}, s_{i_j}) \rightarrow (s_{i_j}, G_j) : 0 \leq j \leq n - 1\}$ is consistent.

A TSGD is said to be strongly-acyclic if it does not contain any strong-cycles. □

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 a 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 $G_i$. □

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 node $G_i \in V$ such that for all transactions $G_j \in V$, for all sites $s_k$, dependency $(G_i, s_k) \rightarrow (s_k, G_j) \notin D$. Also, 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. □

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 $RT$ in $S$ involving $G_i$ could result 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 TSGD 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 involves restricting the execution of $ser_k(G_i)$ operations 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 an $init_i$ operation is processed, but instead, ensures that at any instant the following invariant holds: 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 correctness 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 $init_i$ operations 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 \(s_t\).

The dependencies added to the TSGD during the processing of an \(\text{init}_i\) operation ensure that no 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 \(S\) 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. \(\text{Detect.Ins.TSGD}1\) can be shown to terminate in \(O(n_s n_i^2 m)\) steps and \(\text{Detect.Ins.TSGD}2\) can be shown to terminate in \(O(n_s n_i^3 m)\) steps. Since, in the worst case, \(\text{Detect.Ins.TSGD}7\) is invoked 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 \(O(n_s n_i^2 m n_R v_s)\) if \(\text{Detect.Ins.TSGD}7\) is \(\text{Detect.Ins.TSGD}1\) and is \(O(n_s n_i^3 m n_R v_s)\) if \(\text{Detect.Ins.TSGD}7\) is \(\text{Detect.Ins.TSGD}2\). \(\Box\)

Among the conservative schemes presented in the last two sections, the TSGD scheme with \(\text{Detect.Ins.TSGD}2\) provides the highest degree of concurrency, but also has the highest complexity. The TSG scheme with \(\text{Detect.Ins.TSGD}1\) has the lowest complexity among all the schemes, but also permits the lowest degree of concurrency. The TSGD scheme with \(\text{Detect.Ins.TSGD}1\) and the TSG scheme with \(\text{Detect.Ins.TSGD}2\) have identical complexities, but the degree of concurrency provided by the two 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 \(\Delta' \subset \Delta\) 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 \(\Delta' \subset \Delta\), 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 \(v_0, v_1, \ldots, v_{n-1}\), \(n > 2\), in the TSGD such that \((v_1, v_0)\)\(\rightarrow\)(\(v_0, v_2\)) \(\in D\), \((v_2, v_0)\)\(\rightarrow\)(\(v_0, v_3\)) \(\in D\), \(\ldots\), \((v_{n-1}, v_0)\)\(\rightarrow\)(\(v_0, v_1\)) \(\in D\). \(\Box\)

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 $ser_k(G_i) \in G_i$, for all transactions $G_j \in V$ such that $ser_k(G_j) \in G_j$ and $act(ser_k(G_j))$ has executed, dependencies $(G_j, s_k)\rightarrow(s_k, G_i)$ are added to $D$.

$$\Delta := \emptyset;$$

for every regular term $RT = e_0:reg.exp$ in $R$ such that $\text{type}(G_i) = \text{hdr}(e_0)$ do

for every subtransaction $G_{ik}$ such that $\text{type}(G_{ik}) = \text{last}(e_0)$ do

begin

if $\text{arity}(e_0) = 1$ then $set_1 := \{s_k\}$

else $set_1 := \{s_i : (s_i \neq s_k) \land (\text{type}(G_{ii}) = \text{first}(e_0))\}$;

$$\Delta := \Delta \cup \text{Detect}\_\text{Ins}\_\text{TSGD}((V,E,D \cup \Delta,L),G_{ii},s_k,\text{set}_1,RT)$$

end:

$D := D \cup \Delta$

Figure 9: Pseudocode for $act(init_i)$

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

- $\text{cond}(\text{ack}(ser_k(G_i)))$: true.

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

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

Procedures Detect_Ins_TSGD1 and Detect_Ins_TSGD2 traverse edges in the TSGD in order to detect potential instantiations, and are very similar to procedures Detect_Ins_TSG1 and Detect_Ins_TSG2 respectively. Detect_Ins_TSGD1 and Detect_Ins_TSGD2 are similar to Detect_Ins_TSG1 and Detect_Ins_TSG2, in that they may detect false instantiations (Detect_Ins_TSGD1 detects more false instantiations than Detect_Ins_TSGD2). However, instead of returning a set of site nodes, they return a set of dependencies that if added to the TSGD, restrict the execution of the appropriate $ser_k(G_i)$ operations so that there are no instantiations involving $G_i$.

The updates to $\text{ance}(v')$ and $V.\text{set}(v')$ when a node $v'$ is visited are the same in both Detect_Ins_TSG1 and Detect_Ins_TSGD1. One of 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, Detect_Ins_TSGD1 ensures that for all $i = 1, 2, \ldots, p$, $(v_i, u_i)$ is distinct from $(v_0, u_0)$ (unlike 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, and 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 $\text{ance}(v')$ when $v'$ is visited in state $st$.

Detect_Ins_TSGD2, too, updates $\text{ance}(v')$ and $V.\text{set}(v')$, when a node $v'$ is visited, in a manner identical to Detect_Ins_TSG2, and like 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) and 3(d), for any state $st$ of $F$, every node $v'$ in the TSGD may need to be visited in state $st$ once
in the same state st multiple times, each time the same node u being appended to \( \text{anc}(v') \), \((st, u)\) is added to \(V\text{-set}(v')\) when \(v'\) is visited in state \(st\) and \(u\) is appended to \(\text{anc}(v')\).

Similarly, in 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 \(v'\) 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 \(st\) being 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 \(V\text{-set}(v')\) when \(v'\) is visited in state \(st\) and \((w, u)\) is appended to \(\text{anc}(v')\).

When an \(init_i\) operation is processed, in order to prevent instantiations involving transaction \(G_i\), the TSG scheme restricts the execution of certain \(ser_k(G_i)\) operations by marking them (processing of 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 \(init_i\) 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. □

The complexity of the TSG scheme is dominated by the number of steps it takes to process an \(init_i\) operation. Procedures Detect.\_Ins.\_TSG1 and Detect.\_Ins.\_TSG2 can be shown to terminate in \(O(nS\cdot n_G m)\) and \(O(nS\cdot n_G^2 m)\) steps respectively. Since, in the worst case, when \(init_i\) is processed, 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 \(O(nS\cdot n_G m n_{RV} S)\) if Detect.\_Ins.\_TSG? is Detect.\_Ins.\_TSG1 and is \(O(nS\cdot n_G^2 m n_{RV} S)\) if Detect.\_Ins.\_TSG? is Detect.\_Ins.\_TSG2. □

### 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 \(ser_k(G_i)\) 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 \(G_i\) are detected when the edges of the TSGD are traversed during the processing of an \(init_i\) operation, then appropriate dependencies are added to the TSGD in order to restrict the processing of certain \(ser_k(G_i)\) operations. We now specify, for every operation \(o_j\) in QUEUE, \(cond(o_j)\) and \(act(o_j)\) (no actions are performed when a \(cal_i\) operation is processed, and \(cond(cal_i) = true\)). Initially, for the TSGD, \(V = \emptyset\), \(E = \emptyset\), \(D = \emptyset\).

- **\(cond(init_i)\):** \(true\).
- **\(act(init_i)\):** The pseudocode in Figure 9 is executed. Procedure Detect.\_Ins.\_TSGD? can be either Detect.\_Ins.\_TSGD1 (see Figure 13 in Appendix A) or Detect.\_Ins.\_TSGD2 (see Figure 14 in Appendix A).
- **\(cond(ser_k(G_i))\):** For all transactions \(G_j \in V\), if dependency \((G_j, s_k) \rightarrow (s_k, G_i) \in D\), then \(act(\text{ack}(ser_k(G_j)))\) has completed execution.
\( G_i \) and its edges are inserted into the TSG. Also, for every operation \( \text{ser}_k(G_i) \in G_i \), \( \text{ser}_k(G_i) \) is inserted at the end of the queue for site \( s_k \).

\[
set_2 := \emptyset;
\]
\[
\textbf{for} \text{ every regular term } RT = e_0 : \text{reg.exp} \text{ in } R \text{ such that } type(G_i) = \text{hdr}(e_0) \text{ do}
\]
\[
\textbf{for} \text{ every subtransaction } G_{ik} \text{ such that } type(G_{ik}) = \text{last}(e_0) \text{ do}
\]
\[
\text{begin}
\]
\[
\textbf{if} \text{arity}(e_0) = 1 \text{ then } set_1 := \{s_k\}
\]
\[
\textbf{else} \text{ } set_1 := \{s_l : (s_l \neq s_k) \land (type(G_{il}) = \text{first}(e_0))\};
\]
set_2 := set_2 \cup \text{Detect.Ins.TSG?((V, E, L), G_i, s_k, set_1, set_2, RT)}
\]
\[
\text{end}
\]

For every site \( s_l \) in set_2, \( \text{ser}_l(G_i) \) is marked in the queue for site \( s_l \).

Figure 8: Pseudocode for \( \text{act}(\text{init}_i) \)

- \( \text{act}(\text{fin}_j) \): For every transaction \( G_j \in V \) such that \( \text{val}_j \) 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, \( \text{val}_k \) 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 \( \text{ser}_k(G_i) \) 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 node \( v' \) is visited, the nodes appended to \( \text{anc}(v') \) 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 \( j, k, j = 0, 1, 2, \ldots, n-1, j < k < j+n \), it is the case that for all \( l, j < l < k, \text{arity}(t_{l \mod n}) = 1 \), then \( \text{hdr}(t_j) \neq \text{hdr}(t_{(j+1) \mod n}) \neq \cdots \neq \text{hdr}(t_{k \mod n}) \). 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 an 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 \( (v_0, u_0) \) and \( (v_p, u_p) \) are distinct, while procedure Detect.Ins.TSG2 goes one step further and ensures that for all \( i = 1, \ldots, p, (v_i, u_i) \) is distinct from both \( (v_0, u_0) \) and \( (v_{i-1}, u_{i-1}) \). For this purpose, Detect.Ins.TSG1 appends to \( \text{anc}(v') \), 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 \( \text{anc}(v') \), 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 need to be visited in state \( st \) twice during the execution of Detect.Ins.TSG1, the nodes appended to \( \text{anc}(v') \) the two times \( v' \) is visited in state \( st \) being distinct. In order to prevent a node \( v' \) from being visited
Theorem 1: Let $R$ be a complete regular specification. The optimistic scheme ensures that $S$ is correct with respect to $R$.

Proof: See Appendix B. $\square$

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 $O(n_S n_G^2 m)$ steps. Since, in the worst case, Detect_Ins_Opt is invoked for every regular term in $R$ and for every subtransaction of global transaction $G_i$, the complexity of the optimistic scheme is as stated in the following theorem.

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

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. The schemes utilize a data structure referred to as the transaction-site graph (TSG). A TSG is similar to the TSGD, except that it contains no dependencies. Thus, a TSG is a 3-tuple $(V, E, L)$. Also, associated with every site $s_i$, 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 certain $ser_k(G_i)$ operations 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$_i$.
- $cond(fin_i)$: $true$. 

16
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.

\[
\text{for every regular term } RT = e_0 : \text{reg.exp} \text{ in } R \text{ such that } type(G_i) = hdr(e_0) \text{ do}
\]

\[
\text{for every subtransaction } G_{ik} \text{ such that } type(G_{ik}) = last(e_0) \text{ do}
\]

\[
\begin{align*}
\text{begin} \\
\text{if } arity(e_0) = 1 \text{ then } set_1 := \{s_k\} \\
\text{else } set_1 := \{s_l : s_l \neq s_k \land (type(G_{il}) = \text{first}(e_0))\}; \\
\text{if } \text{Detect.Ins.Opt}(V', E', D', L'), G_i, s_k, set_1, RT = \text{abort} \text{ then}
\end{align*}
\]

\[
\text{begin} \\
\text{Delete } G_i \text{ along with all its edges and dependencies from the TSGD;} \\
\text{Inform GTM}_1 \text{ to abort } G_i; \\
\text{exit}() \\
\text{end} \\
\end{align*}
\]

Inform GTM$_1$ to commit $G_i$

Figure 7: Pseudocode for act(val;)

and element $l_0$, head($l$) returns $l_1$, tail($l$) returns $[l_1, \ldots, l_p]$ and $l_0 \circ l$ returns $[l_0, l_1, l_2, \ldots, l_p]$. Also, for an ordered pair $o = (o_1, o_2)$, $o[0] = o_1$, while $o[2] = o_2$.

Detect.Ins.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.Ins.Opt traverses an edge, 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.Ins.Opt). The current node being visited is stored in variable $v$, while the current state of $F$ is stored in variable cur.st. Since instantiations may contain elements with arity 1, edge traversals do not always result in a new node being visited. If, for an edge $(v, u)$, $st' = st_F(\text{cur.st}, L(v, u))$ is defined, and if on traversal of edge $(v, u)$, cur.st is set to $st'$, then the node visited is $v$ itself (since the traversed edge $(u, v)$ corresponds to an element with arity 1 in the instantiation). We refer to the edge traversal as a 1-arity traversal. However, if for edge $(v, u)$, $st = st_F(\text{cur.st}, L(v, u))$ is defined, and if on traversal of edge $(v, u)$, cur.st is set to $st$, then the node visited is $u$ (thus, edge $(v, u)$ is traversed normally). We refer to the edge traversal as a 2-arity traversal.

Since for any two consecutive elements $t_i$ and $t_{(i+1) \mod n}$ in an instantiation $t_i : t_{i+1} \ldots t_{n-1}$, $hdr(t_i) \neq hdr(t_{(i+1) \mod n})$, and for an element $t_j$ with arity 2, first($t_j$) \neq last($t_j$), consecutive edges traversed by Detect.Ins.Opt must be distinct. This is ensured by appending to the list $\text{last}(v')$, when $v'$ is visited, the node $u$ such that $(u, v')$ 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 the 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 same 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, the current node and the current state of $F$ just before $v'$ was visited is appended to $F.list(v')$ to enable backtracking from $v'$ to take place (Step 4 of procedure Detect.Ins.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 argument on the number of val$_i$ operations processed, it can be shown that global schedule $S$ is correct.