SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 8, Number 2, pp. 197–208. http://www.scpe.org

ISSN 1895-1767

AN EXPERT SYSTEM FOR ANALYSIS OF CONSISTENCY CRITERIA IN CHECKPOINTING ALGORITHMS

SHAHRAM RAHIMI, GURUPRASAD NAGARAJA, LISA GANDY AND BIDYUT GUPTA*

Abstract. In a distributed computing environment, it is vital to maintain the states of the processes involved in order to cater to failures that are arbitrary in nature. To reach a consistent state among all the processes, checkpoints are taken locally by each process and are combined together based on uniformity criteria such as consistency, transitlessness, and strong consistency. In this article, first, the necessary and sufficient conditions of consistency criteria are stated and then an expert system, implemented based on these criteria, is presented. The expert system discovers and illustrates consistent, transitless, strongly consistent and globally consistent checkpoints in a given distributed system. Moreover, it offers facilities for evaluating checkpointing algorithms by measuring different quality assessment parameters.

1. INTRODUCTION. A distributed computing environment consists of a number of processes involved in computation and communicating with each other. In such an environment, there is a need for a mechanism to recover and proceed with the computation, if one or more of the processes fail at any instant of time during computation. Variety of checkpointing and recovery techniques have been proposed (synchronous, asynchronous, hybrid to name a few), in order to minimize re-computing involved in the recovery steps [5, 6, 7]. Generally, recovery includes the rollback of the processes involved in the computation to a point, from where if the computation were to restart, the final result would be the same as that without the failure(s). This is termed as a globally consistent state or a recovery line. In section 2, some background regarding checkpointing and its consistency issues are given.

This paper presents an expert system capable of finding all the possible globally consistent states over a fixed time interval. It also traces consistent, transitless and strongly consistent states between any two or more processes in a distributed system. With these features, the tool may be used for verification of the correctness and efficiency of other checkpointing and recovery algorithms. These algorithms can be checked for their correctness in providing/discovering recovery lines or to see if the consistency criteria are being exposed accurately. Moreover, the system provides facilities for evaluating different algorithms by comparing their features. Currently, the software calculates the following characteristics for a given checkpointing algorithm:

- average number of the checkpoints taken by a process in a given time,
- number of globally consistent checkpoints in a given time,
- average number of checkpoints skipped by a process when rolling back to a recovery line, and
- average elapsed time when rolling back to a recovery line.

To our knowledge, there exists no tool with features matching or even close to the proposed system.

Originally, a C++ program, and not an expert system, was implemented with some of the noted features. The program was extremely slow due to the exhaustive search process for determination of the consistent pairs of the checkpoints. Moreover, implementation of the consistency criteria (based on the theorems, lemmas and definitions discussed in the next section), using a sequential/procedural language such as C++ produced a complex and hard to modify code. Because of these drawbacks, a non-procedural, declarative rule-based engine, Java Expert System Shell (JESS) [4], was employed to develop the system. Using JESS considerably simplified the code, improved the performance in average over four times, and eased the maintenance and upgrade of the system. The reason for these improvements lies under the fact that in a rule-based program, any of the rules may become activated and put on the agenda if its antecedent matches the facts, while the order that the rules were entered does not affect which rules are activated. Thus, the order of the the program statements does not specify a rigid control flow which makes it a logical fit for the framework of the consistency criteria. This is because the consistency criteria are materialized using theorems, lemmas and definitions that could be treated opportunistically.

In section 2, a brief description of a distributed system is given and definitions of consistency, transitlessness and strong consistency are stated. Moreover, methods of finding these criteria in a general graph are explained in this section. In section 3, the architecture of the expert system for the analysis of consistency criteria is presented and its correctness is verified in section 4, using an example. The paper is concluded with a summery and future work section.

2. CONSISTENCY ISSUES IN DISTRIBUTED CHECKPOINTS. Consider a distributed computing environment consisting of N processes that interact with each other by exchange of messages. An event occurs each time a process sends or receives a message. *Lamport's happened-before* relationship is used to define these events. If $a \overset{hb}{\rightarrow} b$ then

*Department of Computer Science, Southern Illinois University, Carbondale, Illinois 62901.

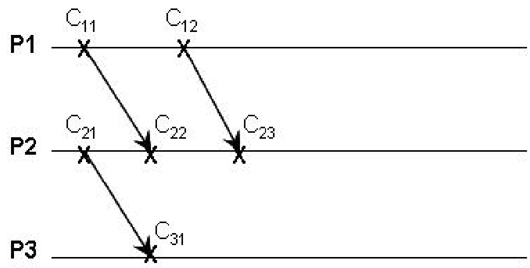

FIG. 1. *Local Checkpoints.*

it is said that event *a* caused event *b*, or a *causal path* exists between *a* and *b* [2]. An example of a *happened-before* relationship is where a process, P_1, sends a message to another procss, P_2. Since the *"send"* event, *a*, from Process P_1 happens before, and is the cause for the *"receive"* event, *b*, at process P_2, it is defined as $a \overset{hb}{\rightarrow} b$ If two events *a* and *b* do not have a *happened-before* relationship between them, then it is said that they are *unordered*, otherwise they are *ordered*.

In a multi-process system, a global state is recorded by combining local checkpoints (periodical snapshots of the processes involved in the computation), one per participating process. In order to group the local checkpoints into a global checkpoint, the necessary and sufficient conditions, proved in [1] are used. A local checkpoint might be taken *synchronously* [5, 11], enabling easy recovery, or *asynchronously* [6], which reduces the number of message exchanges among processes, depending on the preferred algorithm. In this paper, the notation C_{ij} is interpreted as the j^{th} local checkpoint of process P_i. In Fig.1, local checkpoints labeled as C_{11}, C_{11}, C_{21}, C_{22} and C_{31} record corresponding local states of the processes P_1, P_2 and P_3 respectively. If C_{11}, C_{21} and C_{31} are combined together then they define a global checkpoint [2].

These global states play a vital role when one of the processes involved in computation fails and the entire system has to be restored to a state from where the computation can resume without affecting the final result. Therefore the choice of a consistent global state has to be carefully made. In Fig.1, C_{12}, C_{22} and C_{31}, if combined together, constitute a safe global checkpoint in case of the failure of P_1, P_2 and/or P_3. However, C_{11}, C_{21} and C_{31}, if grouped together, do not yield a globally consistent state for recovery. This is because any message sent after the checkpoint C_{21} from P_2, before the checkpoint C_{31} to process P_3, will be lost and produce an incorrect final result.

In [1], Helary describes the transformation of a happened-before relationship to a Z-graph. If a Z-graph exists between two checkpoints, belonging to two different processes, then the checkpoints are not consistent with each other. Another possible transformation of a happened-before relationship could be to a τ-graph used to decide the transitlessness of two checkpoints belonging to two different processes. An S-graph is defined as a union of a τ-graph and a Z-graph and is used to find strongly consistent checkpoints. Z-graph, τ-graph, and S-graph are discussed in detail later in this section.

In this section, the definitions of consistency, transitlessness and strong consistency are reviewed and the necessary and sufficient conditions are stated. However, proofs are considered beyond the scope of this paper.

2.1. Consistency Criterion. A pair of consistent checkpoints [10, 12] should not have any causal path between them. In other words, consistent checkpoints cannot exhibit messages received but not yet sent. That is there cannot be an *orphan* message between any pair of consistent checkpoints. A message *m* sent by a process, P_i, to a process, P_j, is called orphan with respect to the ordered pair of local checkpoints (C_{ix}, C_{jy}) if and only if the delivery of *m* belongs to C_{jy} *(deliver(m)* $\in C_{jy}$*)* while its sending event does not belong to C_{ix} *(send(m)* $\notin C_{ix}$*).* In Fig 2, message m_1 is an orphan message because the sending of message m_1 is not recorded by C_{11} but the receiving of m_1 is recorded by checkpoint C_{21}. Therefore, the ordered pair of local checkpoints (C_{11}, C_{21}) is not consistent. However, the ordered pair of local checkpoints (C_{12}, C_{22}) is consistent due to the absence of any orphan messages. Similarly, the pair of checkpoints (C_{22}, C_{31}) and (C_{12}, C_{31}) are consistent. Together they constitute globally consistent checkpoints (C_{12},C_{22}, and C_{31}).

We can thus define a consistent global checkpoint as follows:

DEFINITION 1: *A global checkpoint is consistent if all its pairs of local checkpoints are consistent.*

2.1.1. Z - Path Instantiation. Definition 1 can be used to transform the graph displayed in Fig. 2 into a Z-graph that helps to detect the Z-paths and therefore eliminate those checkpoints that cannot be considered for global consistency. As [1] enunciates, a graph (as shown in Fig. 2) is said to have a Z-path between two checkpoints C_i and C_j, taken before an event e_i in process P_i and after an event e_j in process P_j respectively, if e_i and e_j are communication events between P_i and P_j and concern the same orphan message *m*.

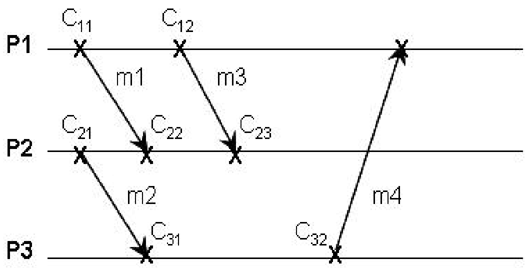

FIG. 2. *m1 is an orphan message between C_{11} and C_{22} and in-transit between C_{12} and C_{21}.*

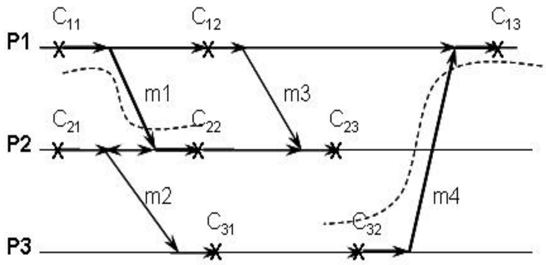

FIG. 3. *A Z-Path between ordered pair of local checkpoints (C_{11}, C_{22}) and (C_{12}, C_{23}).*

Fig. 3 modifies Fig. 2 to demonstrate the existence of the *Z* paths, which are represented by dotted lines. Since the checkpoints C_{11} and C_{22} have a *Z*-Path between them, they cannot participate in a globally consistent checkpoint. For similar reasons, C_{13} and C_{32} cannot participate in a globally consistent checkpoint either.

2.1.2. Necessary and Sufficient Conditions for Consistent Checkpoints. Lemma 1 states that $\sum$ can be a globally consistent checkpoint if and only if there exists no *Z*-Path between any two ordered pair of local checkpoints that are in $\sum$. Similar inferences for a set of local checkpoints *M* and a local checkpoint *C* follows. Note the direction of the arrows decide the nature of the path. In Fig. 3, *Z*-path exists between C_{11} and C_{22} and is depicted using dotted lines.

LEMMA 1. A global checkpoint $\sum$ is consistent if and only if $\neg(\sum \overset{Z}{\rightarrow} \sum)$.

THEOREM 1. *Let M be a set of local checkpoints from different processes. M can be extended to a consistent global checkpoint if an only if $\neg(M \overset{Z}{\rightarrow} M)$.*

COROLLARY 1. *Let C be a local checkpoint. C can be a member of consistent global checkpoint if an only if $\neg(C \overset{Z}{\rightarrow} C)$.*

2.2. Transitless Criterion. Transitless checkpoints cannot exhibit messages sent but not yet received and therefore are the dual opposites of the consistent checkpoints explained in section 2.1. As we can see, this condition suggests that there cannot be any message *in-transit* for an ordered pair of checkpoints to be transitless. It may include messages received but not yet sent; in such cases, the checkpoints can be only transitless and not strongly consistent. This is explained further in section 2.3. Message *m* sent by process P_i to process P_j is said to be in-transit with respect to the ordered pair of checkpoints (C_{ix}, C_{jy}) if and only if the sending of *m* belongs to C_{ix} (*send(m)* $\in C_{ix}$) while the receiving of *m* does not belong to C_{iy} (*receive(m)* $\notin C_{iy}$). In Figure 2, the ordered pair (C_{12}, C_{21}) is an ordered pair of checkpoints that have message *m1* in-transit. This is because the checkpoint C_{12} records the sending of message *m1* while C_{21} does not record the receiving of the message *m1*. However the ordered pair (C_{12}, C_{22}) is transitless as it does not involve any messages in-transit.

DEFINITION 2: *A global checkpoint is transitless if all its pairs of local checkpoints are transitless.*

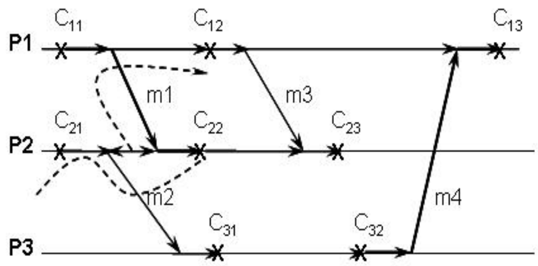

FIG. 4. *τ-path exists between ordered pair of checkpoints (C_{12}, C_{21}).*

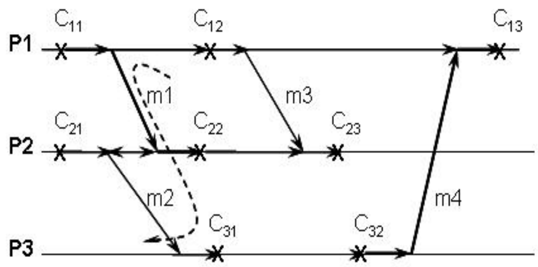

FIG. 5. *An S- Path between C_{12}, C_{22}, C_{31}.*

In Fig. 2, Checkpoints C_{12}, C_{22}, and C_{31} constitute a globally transitless checkpoint since the ordered pairs (C_{12}, C_{22}), (C_{22}, C_{31}), and (C_{12},C_{31}) are all transitless.

2.2.1. τ - Path Instantiation. The idea of having no in-transit messages can be extended to a τ-Path. Assuming that a checkpoint C_i is taken before event e_i in process P_i and checkpoint C_j is taken after event e_j in process P_j, where e_i and e_j are communication events between P_i and P_j and concern the same message, *m*, which is in-transit between P_i and P_j (i. e., events e_i and e_j yield a happened-before relationship which is also called a c-edge), then the graph (as shown in Fig. 2) is said to have a τ-path between the two checkpoints C_i and C_j. In other words, a τ-path exists if there is any in-transit message between two checkpoints. Fig. 4 modifies Fig. 2 to show the existence of τ-path using dotted lines.

2.2.2. Necessary and Sufficient Conditions for Transitless Global Checkpoint. Theorem 2 states that *M* can be a transitless global checkpoint if and only if there exists no τ-path between any two ordered pair of local checkpoints that are in *M*.

THEOREM 2. *Let M be a set of local checkpoints that belong to different processes. M can be extended to a transitless global checkpoint if an only if* $\neg(M \overset{\tau}{\rightsquigarrow} M)$.

2.3. Strong Consistency Criterion. A strongly consistent global checkpoint is made up of local checkpoints that are both consistent and transitless [1]. For example in Fig. 2 local checkpoints C_{12}, C_{22} and C_{31} make up a global checkpoint that is both consistent and transitless; therefore, C_{12}, C_{22}, and C_{31} are strongly consistent.

DEFINITION 3: *A global checkpoint is strongly consistent if all its pairs of local checkpoints are consistent and transitless.*

2.3.1. S-Path Instantiation. An *S*-path is the union of a *Z*-path and a τ-path [1]. Fig. 5 displays an *S*-path that exists between C_{12}, C_{21} and C_{31} and therefore, do not form a strongly consistent global checkpoint. However, C_{12}, C_{22} and C_{31} constitute a strongly consistent checkpoint due to the absence of a *S* path between them.

2.3.2. Necessary and Sufficient Conditions for Strongly Consistent Global Checkpoint. Theorem 3 states that *M* can be a strongly consistent global checkpoint if and only if there exists no *S*-path between any two ordered pair of local checkpoints existing in *M*.

```
p1:send,p2,m1,5:recv,p2,m2,9:send,p2,m5,9:recv,p3,m7,20
p2:recv,p1,m1,7:send,p1,m2,5:recv,p3,m3,3:send,p3,m4,6:recv,p1,m5,7:send,p3,m6,5
p3:send,p2,m3,10:recv,p2,m4,14:recv,p2,m6,11:send,p1,m7,5
```

FIG. 6. *Example of an input file.*

```
Process Name :< send/recv >, < Other Process >, < Message Name Tag>, < Time Elapsed >:
```

FIG. 7. *Structure of a single line of the input file.*

THEOREM 3. Let *M be a set of local checkpoints from different processes. M can be extended to a strongly consistent global checkpoint if an only if* $\neg(M \overset{S}{\rightarrow} M)$ [1].

3. THE EXPERT SYSTEM IMPLEMENTATION. Based on the definitions and theorems as well as the necessary and sufficient conditions described in section 2, an expert system was implemented to determine consistent, transitless, strongly consistent, and globally consistent checkpoints in a distributed environment. Moreover, some features for evaluation purposes were included, such as determining the average number of checkpoints taken by a process, the number of globally consistent checkpoints in a time interval, and the number of messages sent and received for checkpointing purposes.

The application is implemented in Java using the Java Expert System Shell (JESS) [4]. JESS is the Java version of CLIPS (C Language Integrated Production System) [8]. The rule base of the expert system is created from rules that determine various consistency criteria. A snapshot of the distributed system containing the time of the sent and the received messages and the times of the checkpoints taken by each process involved in the computation is fed to the expert system as a set of facts (input). On execution, the facts are evaluated against the rule base to determine the consistency criteria. This section presents the structure of the expert system by discussing its various components at a greater detail.

3.1. Input and Display of Events. The presented expert system takes an ordered set of events, with respect to each process, as its input. Fig. 6 illustrates a sample input file while Fig. 7 describes the generic structure of the contents of the file consisting of send and receive events for a single process.

Each process's event, presented in the input file, has four elements. The first element of each event is the event type denoted by *send*, for a sent message, and *recv*, for a received message. The second element is the name of the process to which a message is sent or from which a message is received; the third element is the name tag of the message. Finally, the estimated time at which the send or receive events occurs is given. Time is represented by generic unit and it is up to the user to decide the representation that is most useful; time is calculated not from the initiation of a process but from the execution of the last event.

The file is then parsed and a graphical display of the communication events between the processes, as specified in the input file, is demonstrated with arrows indicating the *send* and the *recv* events (Figure 8). Also, as the input file is parsed, the local checkpoints are depicted based on the checkpointing scheme employed in the system. This is done through the use of another input file called *checkpointing file*, which is formed either manually by the user or by the processes involved in the distributed computation. Each line of this file represents the estimated time of checkpoints taken by a particular process. In this paper, we have assumed that the checkpoints are taken before the *send* and after the *recv* events.

The Java implementation consists of several classes, but the most important ones are *All_Processes* and *Process*. The *All_Processes* class has a java defined Vector of Process object. When the program begins execution the main method of the class *Checkpoints'* is called. The main method instantiates a *CheckpointFrame* object which then instantiates a *DrawingPanel* object. The *DrawingPanel* object overrides the *paintComponent* method of the *JPanel* class. The *paintComponent* method is where all the drawing to the *JPanel* is done. Inside the *paintComponent* method the *parseFile* method of the *All_Processes* class is called. This is an important method that parses through the given input files and builds *N* process objects, where *N* is the number of user defined Processes. Each Process object has an *events'* vector, an *eventCoords* vector, and a *name*. The name is taken from the input file (i. e. for Fig. 6 the names of the processes would be P_1, P_2, and P_3). The events are also taken from the given input file, and each event is added to the *Process* class's

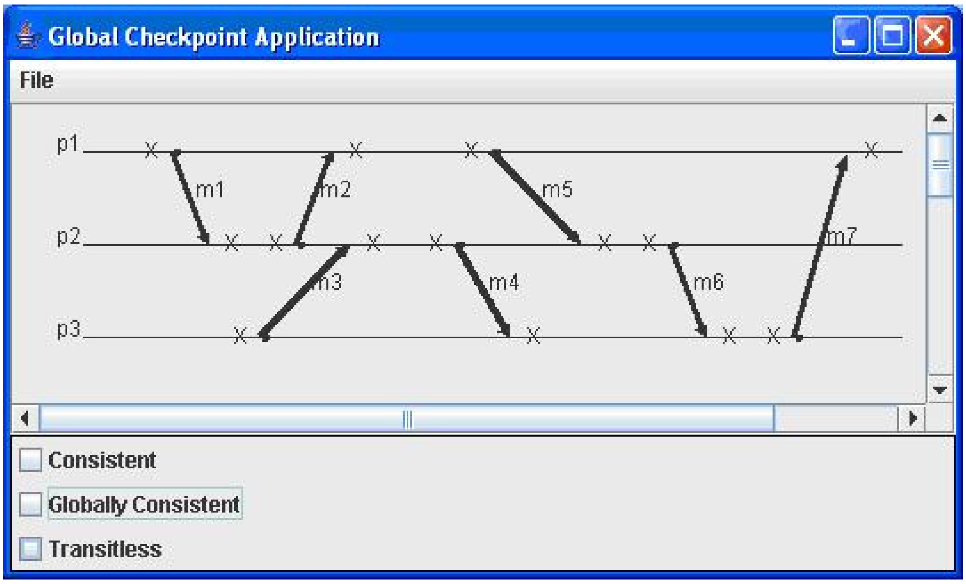

FIG. 8. *Illustration of the events and checkpoints of a distributed system consisting of three processes.*

vector. Once the events are read in, then the coordinates of each event are defined (discussed later in this section) and then are added to the *Process* class's vector. Currently ten pixels are drawn for every time unit that the user defines. So if the user puts in a 7 then 70 pixels are drawn from the last event to the current event. Moreover, as the input file is parsed, local checkpoints are added to the events and *eventCoords* vector, by reading in the checkpointing file.

Once the *All_Processes' parseFile* method is completed, the control returns to the *paintComponent* method of the *DrawingPanel* class. The *paintComponent* method then uses the methods of *All_Processes* to examine each *Process* object and its events and *eventsCoords* vector. The directed graph is drawn from the information given in the events and *eventsCoords* methods of each Process object. The display depicted from the input file, shown in Fig. 6, is exhibited in Fig. 8.

3.2. Converting the Events to JESS Facts. The input file is further interpreted in Java to produce *vector points*, one vector point for each process. For instance, process P_i is assigned vector point V_i which consists of *N* coordinates for *N* processes involved in the distributed system. The concept of vector clocks [9] is modified and utilized to assign values to these vector points. The modified vector clock algorithm facilitates tracking concurrent events among processes and therefore helps the expert system to apply the consistency criteria.

To further describe the vector points, a system with three processes involved in mutual communication is considered in the following example. Since there are three processes involved, the vector point of process P_i, V_i, consists of three coordinates, (V_{i1}, V_{i2}, V_{i3}). Coordinate V_{ij} acts as a counter that keeps track of the number of *send* and *recv* events of process P_j for process P_i. Following are the rules used to assign values to each vector point, which, as was mentioned before, is a modified version of vector clock algorithm.

VC1: Initially, all clocks are 0 on all components. *VC2*: P_i sets V_i[i] := V_i [i] + 1 just before time stamping an event. *VC3*: P_i includes t = V_i in every message it sends to the other processes. *VC4*: P_i receives a timestamp t from P_j, and then computes: *Vi[j] := max(*V_i*[j], t[j])*

The only modification to vector clock algorithm is done for rule *VC4*. In the original vector clock algorithm, *Vi[j] := max(Vi[j], t[j])* is executed for *j = 1 to N*. However, in the modified version, it is executed only for process P_j coordinate from which P_i is receiving the message. This is because of the importance of the pair wise evaluation of the checkpoints for consistency and transitless evaluations in the rule base, which makes the foundation for other evaluations as well. Fig. 9 displays the vector points for the events displayed in Fig. 8.

The vector points then are asserted directly as facts to JESS to be used to determine the pairs of consistent and transitless checkpoints. Fig. 10 illustrates the code that accomplishes the assertion task. These facts are the direct translations of the vector points displayed in Fig. 9. They are then executed against JESS consistency and transitlessness rules that are explained in sections 3.2 and 3.3. As an example, the fact for the vector point $\langle 100 \rangle$ in process P_1 would be (*point (process 1)(coordinates 1 0 0)(index 0)*).

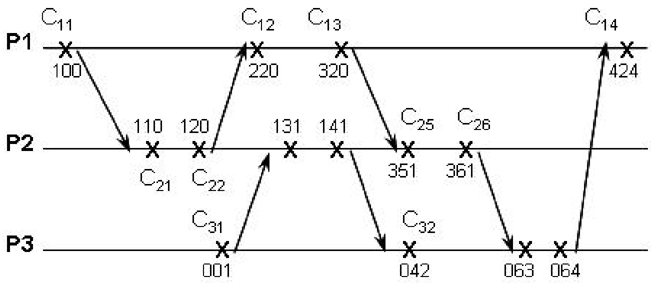

FIG. 9. *Vector points formed using a modified vector clock algorithm.*

```
Rete r = new Rete();
r.clear();
String command = "(batch \"" + myPath3 +        /prjrules/consistent.CLP\")";
r.executeCommand(command);
Deftemplate template = r.findDeftemplate("point");
int I;
for (i=0;i<this.getNumProcesses();i++){
            Process p = this.getProcessAt(i);
            Vector eventVector = p.getVectorEvents();
        for (int j=0;j<eventVector.size();j++)
        {
                Fact myFact = new Fact(template);
                nt name = Integer.parseInt(p.getName().subString(1));
                myFact.setSlotValue("process",new Value (name,RU.INTEGER));
                String eventString = (String)eventVector.elementAt(j);
                ring[ ] eventStringArr = eventString.split(":");
                Integer trialInt = Integer.valueOf(eventStringArr[1]);
                yFact.setSlotValue("x",new  Vaue(trialInt.intValue(),RU.INTEGER));
                ValueVector vv = new ValueVector(); //set multislot points value
                for (int k=1; k<eventStringArr.length; k++) {
                        vv.add(new Value(Integer.parseInt(eventStringArr[k]), RU.INTEGER));
                }
                myFact.setSlotValue("coordinates ", new Value(vv, RU.LIST));
                myFact.setSlotValue("index",new Value(j,RU.INTEGER));
                r.assertFact(myFact, r.getGlobalContext());
        }
} //end of for loop to assert facts
```

FIG. 10. *Asserting vector points as facts into the expert system.*

3.3. Mechanism for Consistency Criterion. Once the vector points are asserted as facts, the expert system checks them against the rule-base and forms sets of consistent checkpoints. For instance, while dealing with processes P_1 and P_2, our rule states that if the first coordinate for P_1 vector point is greater than that of P_2 and the 2nd coordinate for P_2 vector point is greater than that of P_1 then the vector points are consistent. Likewise for processes P_2 and P_3, we test to see if the 2nd coordinate for P_2 vector point is greater than that of P_3 and the 3rd coordinate for P_3 vector point is greater than that of P_2 then the two points are consistent. The pattern for processes *n* and *m* is that if the n^{th} coordinate for process *n* is greater than that of process *m* and the m^{th} coordinate of process *m* is greater than that of process *n* then the n^{th} and m^{th} processes share consistent checkpoints. The consistent vector points are asserted as a new fact in the form of:
(deftemplate consistent (slot process1) (slot index1) (slot process2) (slot index2))

```
(defrule consistent-rule
  (point (process ?p1) (elements $?points1) (index ?idx1))
  (point (process ?p2) (elements $?points2) (index ?idx2))
  =>
  (if (< ?p1 ?p2)   then
    (bind ?tempPt1A (nth$ ?p1 $?points1))
    (bind ?tempPt2A (nth$ ?p1 $?points2))
    (bind ?tempPt1B (nth$ ?p2 $?points1))
    (bind ?tempPt2B (nth$ ?p2 $?points2))
    (if (and(> ?tempPt1A ?tempPt2A)(< ?tempPt1B tempPt2B))  then
     (assert (consistent (process1 ?p1) (index1 ?idx1)
     (process2 ?p2) (index2 ?idx2))))))
```

FIG. 11. *The expert system rule for finding consistent local checkpoints developed in JESS.*

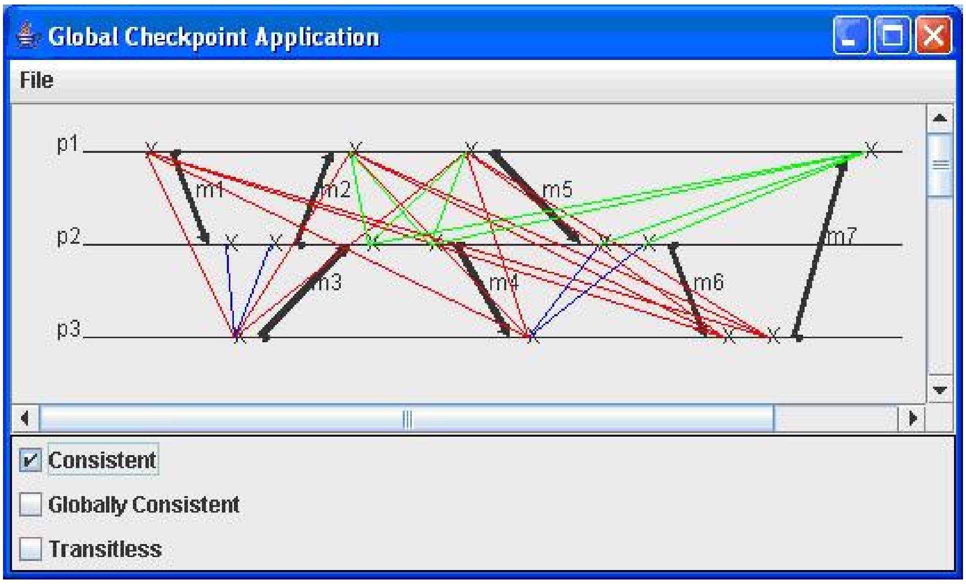

FIG. 12. *Locally consistent checkpoints; different colors indicate consistency between checkpoints of different process pairs.*

Interpretation of the above fact template for consistency rules is as follows: Vector points of Process 1 and Process 2, determined by index1 and index2 respectively, constitute a pair of consistent checkpoints. The expert system rule for determining a pair of consistent checkpoints between any pair of processes in a distributed environment of *n* processes is given in Fig. 11.

A snapshot of the output of the application, displaying the consistent checkpoints between every ordered pair of participating processes, is given in Fig. 12. The program utilizes a color convention for assigning different colors for different pairs of processes. In Fig. 12, system has selected green for consistent pairs between process 1 and process 2, blue for consistent pairs between process 2 and process 3, and red for ordered pairs between process 1 and process 3.

3.4. Mechanism for Transitlessness Evaluation. Once the vector points are asserted as facts, the expert system transitlessness evaluation rule forms sets of transitless checkpoints. When dealing with processes P_1 and P_2, the rule states that if the 1st coordinate for P_1 vector point is greater than that of P_2, then the vector points are transitless. Likewise for processes P_2 and P_3 the rule tests to see if the 2^{nd} coordinate for process P_2 is greater than that of process P_3, and if so, then the two points are transitless. The pattern for any process *n* and process *m* is that if the n^{th} coordinate of process *n* vector point is greater than the m^{th} coordinate for process *m* then process *n* and *m* share a transitless checkpoint. The transitless vector points are then asserted as a new fact in the form of:
(deftemplate transitless (slot process1) (slot index1) (slot process2) (slot index2))

The rule for transitlessness between any two processes' checkpoints among *N* processes is given in Fig. 13. A snapshot of the Java application displaying the resulting transitless checkpoints is given in Fig. 14.

3.5. Mechanism for Strong Consistency Evaluation. Based on DEFINITION 3, strong consistency occurs when checkpoints satisfy both the transitless and consistency criteria. The algorithms to find transitless and consistent check-

```
(defrule transitless-rule
  (point (process ?p1) (elements $?points1) (index ?idx1))
  (point (process ?p2) (elements $?points2) (index ?idx2))
  =>
  (if (< ?p1 ?p2)        then
    (bind ?tempPt1A (nth$ ?p1 $?points1))
    (bind ?tempPt2A (nth$ ?p1 $?points2))
    (if (>= ?tempPt1A ?tempPt2A)
    then
      (assert (transitless (process1 ?p1) (index1 ?idx1)
      (process2 ?p2) (index2 ?idx2))))))
```

FIG. 13. *The expert system rule for finding transitless local checkpoints.*

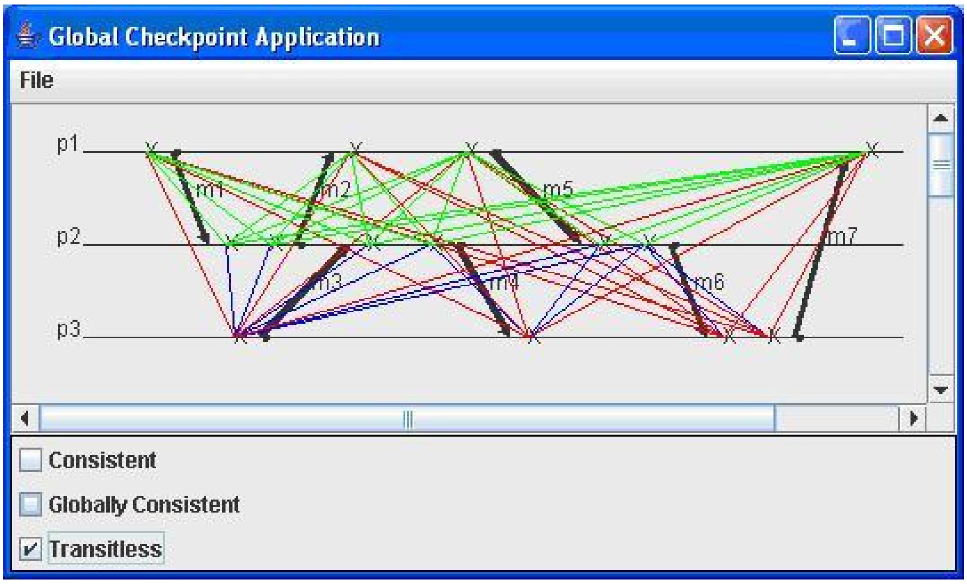

FIG. 14. *Locally transitless checkpoints.*

points are executed and then matching checkpoints are searched for. If the algorithms for transitlessness and consistent checkpoints have the same checkpoints then those checkpoints are considered strongly consistent. Therefore, all checkpoints that are found to be consistent and transitless will be displayed as strongly consistent. An example of the application finding strongly consistent checkpoints is shown in Fig. 15.

3.6. Mechanism for Global Consistency. Globally Consistent checkpoints are composed of local consistent checkpoints (DEFINITION 1). Once the vector points are asserted as facts, the expert system determines the locally consistent checkpoints, as explained in section 3.3, and then checks the set of locally consistent checkpoints against the rule base to determine globally consistent checkpoints. Only the complete sets of local checkpoints that include one local checkpoint per process and in which every pair of the local checkpoints is consistent are retained (THEOREM 1). The rule responsible for finding global consistent checkpoint is assigned a lower salience and therefore is executed after the execution of the rule for consistent local checkpoints. Fig 16 displays the global consistencies in the given distributed system.

4. VERIFICATION. Since the presented expert system was developed based on the thermos, definitions and lemmas presented in section 2 and proven in [1]; therefore, theoretically, it should perform accurately. However, to further verify the accuracy of the system, one hundred randomly formed distributed systems, with 50 processes in each, were generated to evaluate the correctness of the expert system. In these randomly generated distributed systems, the average number of messages set by each process, during the lifetime of the systems, was set to 20 messages, while the average number of the processes that each process communicated with was set to 10 (20 percent of the total number of the processes in each system). The expert system produced accurate results for all of these cases.

In the rest of this section, we consider the example shown in Fig. 9 to verify the expert system capability to trace consistent, transitless and strongly consistent global checkpoints.

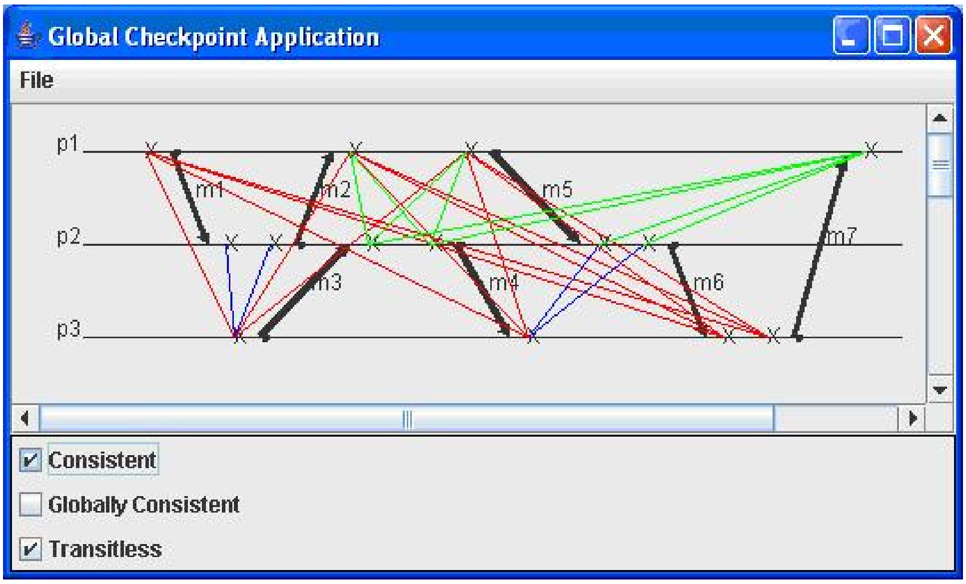

FIG. 15. *Strongly Consistent Checkpoints.*

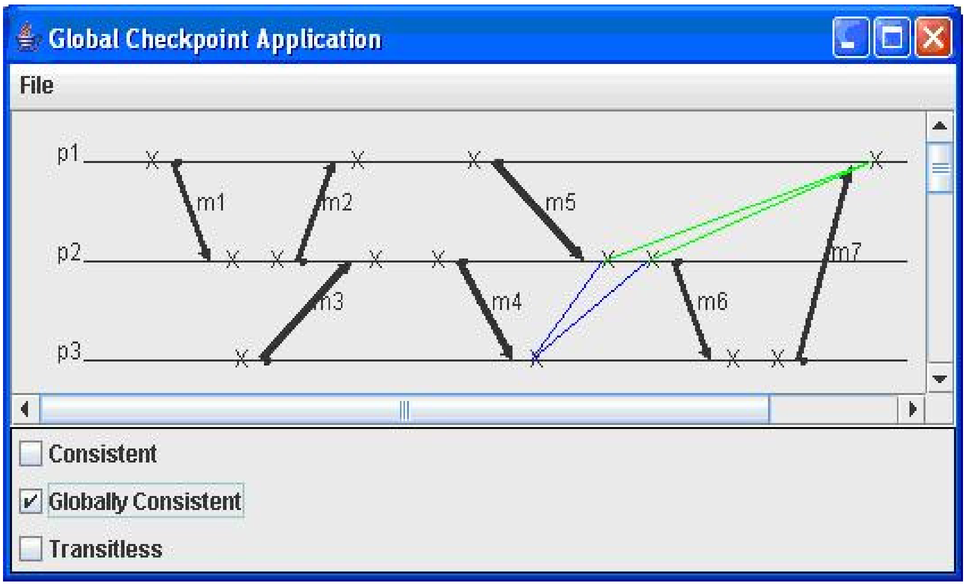

FIG. 16. *Globally Consistent Checkpoints.*

4.1. Consistent Checkpoints. In this subsection, using Fig. 9, we examine the values assigned to the vector points, corresponding to every local checkpoint, and observe the way these values influence the determination of consistent checkpoints. Consider the ordered pair of local checkpoints (C_{11}, C_{21}), with coordinates ({1,0,0},{1,1,0}), corresponding to process P_1 vector point (V_{11}, V_{12}, V_{13}) and process P_2 vector point (V_{21}, V_{22}, V_{23}) respectively. The JESS rule for consistent checkpoints compares the V_{i1} and V_{i2} coordinates as was described before. Since the V_{11} coordinate of C_{11} is not less than the V_{21} coordinate of C_{21}, the ordered pair (C_{11}, C_{21}) is not asserted as consistent checkpoint. For the ordered pair of local checkpoints (C_{21}, C_{31}), the rule in JESS will compare the V_{i2}, V_{i3} coordinates of ({1,2,0}, {0,0,1}). This satisfies the conditions of the rule for consistent checkpoints because the V_{22} coordinate of C_{21} is greater than that of C_{31} and the V_{23} coordinate of C_{21} is less than that of C_{31}; therefore, a fact that says the ordered pair (C_{21}, C_{31}) is consistent is asserted. Fig. 12 is a screen shot of all possible consistent local checkpoints. Finding such pairs for all the processes will yield to the globally consistent checkpoints described in section 3.5.

The assignment of the coordinate values (vector points) for each checkpoint is done in such a way that it eliminates all the checkpoints that are not consistent and mark only those that are consistent. This satisfies the necessary condition that no ordered pair of checkpoints in a globally consistent checkpoint should have a Z-path between them.

4.2. Transitless Checkpoints. For the determination of transitless checkpoints, a similar procedure of comparing the respective coordinates of checkpoints in an ordered pair is followed, depending on which pair of processes is chosen. In the ordered pair of local checkpoints (C_{11}, C_{21}), the coordinates are ($\{1,0,0\}$,$\{1,1,0\}$). Since the receiving of message m_1 is recorded in C_{21}, the ordered pair is transitless. The transitless rule will now check for the V_{11} coordinate of C_{11} to be greater than V_{21} of C_{21}, since this is satisfied, (C_{11}, C_{21}) is identified as transitless.

The assignment of the coordinate values for each checkpoint is done is such a way that JESS rules filters the pairs that are not transitless. Finding such pairs to cover all the processes involved in the computation results in a globally transitless checkpoint. Checkpoints that are consistent and transitless are determined as strongly consistent checkpoints by the system.

4.3. Globally Consistent Checkpoints. The determination of globally consistent checkpoints is carried out in two steps; firstly, determination of locally consistent checkpoints, and secondly, looking for sets of locally consistent checkpoints that include at least one checkpoint per participating process. Extending the verification procedure explained in sections 4.1 determines that the ordered pairs of local checkpoints namely (C_{14}, C_{25}), (C_{25}, C_{32}) and (C_{32}, C_{14}) are locally consistent. Now from DEFINITION 1, we know that a set of checkpoints, if all of its pairs are consistent, becomes a globally consistent checkpoint given that there exists a single checkpoint in the set for every process in the system. In the above example, the three checkpoint pairs are consistent, and every process in the system has a checkpoint participated in the pairs. Therefore, they form a global checkpoint as the expert system accurately detects. Following a similar procedure the expert system traces all possible globally consistent checkpoints.

5. Conclusion and Future Work. The importance of fault-tolerant distributed and grid computing has attracted many researchers to this area. Different checkpointing methodologies, as cost effective solutions for system recovery, have been introduced for many year. This work presents an expert system that could be utilized for evaluating the correctness of various checkpointing algorithms by detecting consistent, transitless, strongly consistent and globally consistent checkpoints produced by recovery algorithms. Moreover, the expert system is capable of comparing features of checkpointing algorithms by calculating, in a given time window, the average number of the checkpoints taken by a process, the number of globally consistent checkpoints, the average number of checkpoints skipped by a process when rolling back to a recovery line, and the average elapsed time when rolling back to a recovery line. It can also help to discover if a checkpointing algorithm is suffering from domino Effect.

Currently new features are being added to the system one of which is to allow processes to supply their checkpointing and message transmission data, in real time, so the determination of the consistency criteria is performed dynamically. The expert system would also need to accommodate dynamic inclusion and exclusion of participating processes in the distributed environment. We claim the presented expert system makes a considerable contribution to research in fault-tolerant distributed computing by serving as an evaluator and a test-bed for checkpointing algorithms.

REFERENCES

[1] HELARY, J. M., AND NETZER, R. H. B., *Consistency Issues in Distributed Checkpoints*, IEEE Transactions on Software Engineering, vol. 25, no. 2, pp 274–281, March/April 1999.

[2] MANIVANNAN, D. ET AL., *Finding Consistent Global Checkpoints in a Distributed Computation*, IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 6, (June 1997), pp. 623–627.

[3] NETZER, R. H. B, AND XU, JIAN, *Necessary and Sufficient Conditions for Consistent Global Snapshots*, IEEE Transactions on Parallel and Distributed Systems, vol. 6, no. 2, Feburary 1995, pp 165–169.

[4] *JESSTM, the Rule Engine for JavaTM Platform*, `http://herzberg.ca.sandia.gov/jess/docs/70/table_of_contents.html` Retrieved November 2004.

[5] CAO, J., JIA, W., JIA, X., AND CHEUNG, T, *Design and Analysis of an Efficient Algorithm for Coordinated Checkpointing in Distributed Systems*, Advances in Parallel and Distributed Computing, Proceedings, (19-21 March 1997), pp 261–268.

[6] MANIVANNAN, D., AND SINGHAL, M, *Quasi-Synchronous Checkpointing: Models, Characterization, and Classification*, IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 7, July 1999, pp. 703–713.

[7] ALVISI, L., ELNOZAHY, E., RAO, S., HUSAIN, S. A., AND DE MEL, A., *An Analysis of Communication Induced Checkpointing*, Fault-Tolerant Computing, Digest of Papers. Twenty-Ninth Annual International Symposium, 15-18 June 1999, pp 242–249.

[8] GIRRATANO, J. AND RILEY, G., *Expert Systems Principles and Programming (Third Edition)*, Boston: Course Technology, Inc.

[9] BALDONI, R. AND RAYNAL, M., *Fundamentals of distributed computing: A practical tour of vector clock systems*, IEEE Distributed Systems Online, vol. 3, no. 2, Feburary 2002.

[10] Elnozahy, E. N.; Johnson, D. B. and Zwaenepoel, W., *The performance of consistent checkpointing*, IEEE Proceedings on Reliable Distributed Systems, 5-7 October 1992, pp 39–47.
[11] Neogy, S., Sinha, A. and Das, P. K., *Checkpoint processing in distributed systems software using synchronized clocks*, IEEE Proceedings on Information Technology: Coding and Computing, 2–4 April 2001, pp 555 - 559.
[12] Manabe, Y.,*A distributed consistent global checkpoint algorithm with a minimum number of checkpoints*, IEEE Proceedings of the Twelfth International Conference on Information Networking, 21–23 January 1998, pp 549–554.

Edited by: Marcin Paprzycki
Received: December 14, 2005
Accepted: October 24, 2006