International Journal of Engineering Trends and Technology
ISSN: 2231 - 5381 /doi:10.14445/22315381/IJETT-V69I4P201
Volume 69 Issue 4, 1-9, April 2021

Formal Specification & Verification of Checkpoint Algorithm for Distributed Systems using Event - B

Bal Krishna Saraswat*[1], Raghuraj Suryavanshi[2], and Divakar Yadav[3]

[1]Assistant Professor, Department of Computer Science & Engineering, SRM Institute of Science & Technology, NCR Campus, Modinagar, India
[2]Assistant Professor, Department of Computer Science & Engineering, Pranveer Singh Institue of Technology, Kanpur, India
[3]Professor, Department of Computer Science & Engineering, Institute of Engineering & Technology, Lucknow, India
[1]*saraswat.banti@gmail.com*, [2]raghuraj.suryavanshi@gmail.com, [3]divakar_yadav@rediffmail.com

***Abstract*—Using formal methods to design a system model, and then specifying and verifying critical properties of that model is a way to design safety critical systems. Modeling can be done by a proper methodology so that one can analyze proposed behavior of the models quantitatively. Formal method used to develop the distributed system models is Event - B. This approach consists of meticulously defining the problem in a conceptual model, incorporating problem solutions or design information in the refinement steps for the sake of obtaining concrete requirements, and checking the accuracy of explanations offered. The existing B tools offer substantial automatic proof assistance to generate and discharge the proof obligations. The various processes at different locations are linked by the network in a distributed system. They communicate with each other by sending messages. A checkpoint is a process's saved local state. If the global state created by the saved states is consistent, a collection of checkpoints, one per system process, is consistent. In this paper a refined methodology is introduced for the development of distributed system models using Event-B, in which processes coordinate their checkpoint through broadcasting messages to always create a consistent set of checkpoints. A formal logic method is needed to understand the behavior of these techniques and achieve the goals.**

***Index Terms*—Event-B, Formal Verification, Distributed systems, Recovery, Checkpoint, Formal Specifications, tentative checkpoint number, permanent checkpoint number, Formal Methods.**

I. Introduction

Over the last two decades, the computing industry has shifted toward distributed, low-cost, high-volume unit goods. With the incorporation of multi-computing and multiprocessing, the computer system's efficiency has significantly improved. The distributed framework has become popular and relevant with the advent of web and network technologies [18]. In every application area, one can find Distributed system including client/server systems, World Wide Web, transaction processing, banking, financial services, and many others.

This paper is about testing system design using modeling of the system. Distributed system's reliability is an important design requirement for improving current distributed systems or designing the new ones. Reliability refers both to a system's vulnerability to different kinds of failures and its ability to survive from them [24]. A system can be configured to be fault tolerant by demonstrating rigorous behavior that encourages the recovery-friendly action [17]. Checkpoint and rollback recovery are well-acknowledged strategies for addressing distributed systems reliability [21], [22], [8]. A checkpoint is an execution's saved intermediate state and can be used to restart the execution from that stage. Existing algorithms for checkpoints can be divided into three classes.

- Uncoordinated (asynchronous),
- Coordinated (synchronous) and
- Communication Induced (quasi-synchronous).

Under asynchronous checkpoint, processes periodically take checkpoints (i.e., the system state is stored periodically in stable storage) [18] without any synchronization with other processes. A consistent global checkpoint [20] is accomplished when a process fails and the processes starts again from such latest consistent global checkpoint. This approach allows checkpoints to be taken whenever process wants, and therefore processes can design the checkpoint operation whenever the I/O nodes are not busy, resulting in fewer checkpoint overhead[15]. Under certain circumstances,

rollback propagation takes place until the start of the computation and entire computation was lost what had takes place before the failure, this is known as domino effect [25].

Synchronized checkpoint is a solution to the issue of uncoordinated checkpoint, in which processes organize their checkpoint operation in synchronous or coordinated checkpoint schemes so that the system always retains a globally consistent set of checkpoints [13].

There is yet one more technique to checkpoint, known as communication induced checkpoint [26], [21]. Under communication-induced checkpoint, the processes are permitted to take the checkpoint on the basis of the information sent with the application messages by the communicating processes. Checkpoints are conducted in a certain manner that there is system-wide consistent state at all the time.

For distributed systems, the correctness of a checkpoint process must be formally verified to ensure fault tolerance of the safety critical systems. In this paper, authors presented the formal development of a checkpoint process that ensures a system wide consistent global state [20] in distributed system. The approach to the step by step evolution of the checkpoint process is established on the abstraction and refinement methodology. This technique is noteworthy for its ability to formally explain abstract global models of a system, and then refine it in a series of intermediate steps to a comprehensive distributed architecture. Event-B [2], [23] supports this methodology, that is an alternative to B Method [4]. Event-B encourages a step-by-step implementation in the refinement phases from the initial abstract specifications. The compliance between the system's development and the abstract requirements are checked at each refinement step. B tools make available for use provide the substantial automatic proof assistance to generate and discharge the proof obligations for consistency and refinement checking. Various B tools support this technique, for instance Rodin [6], B4Free [14], which provides substantial automatic assistance for the definition of proof obligations, factorizing and discharging complicated proof obligations into simplified proofs. The critical characteristics of design, and proof guidelines for achieving a high level of automatic proofs for an Event-B implementation are defined in [12].

In this paper the system development is started with the incremental development of the Distributed system which uses Coordinated Checkpoint [19] using Event - B [3], [9], [10]. Event-B is an event driven methodology utilized to formulate formal models of distributed systems. In the system's abstract model, an abstract distributed system is defined that uses passing message to interact with each other. In the refinement steps it is outlined that how a system can properly implements a Coordinated Checkpoint methodology to achieve fault tolerance.

II. Materials & Methods

This section outlines the informal requirements of the coordinated checkpoint properties and incorporates the Event-B method.

A. Event-B

Event-B [2], [7] is a process modeling technique which follows B method for the design of distributed systems. Operations in Event-B are known as events that takes place impromptu instead of being invoked randomly. Guarding of the events are done by predicates, and at every step of refinement these guards may be strengthened. The variables of the state are changed by events. Invariants states the properties that variable must satisfy and retain through event activation. The application of set theory notation to modelling, the use of refinement to define structures at different levels of abstraction, and the use of mathematical proof to check consistency between levels of refinement are all main features of Event-B. Two fundamental concepts characterize the Event-B models: context and machine [16]. Context comprises a model's static, while machine comprises the dynamic part.

Variables, invariants, events, and variants can be found in machines, while carrier sets, constants, and axioms can be found in contexts. Machines and contexts have distinct names. There are different relationships between machines and contexts: A machine can be refined by some other machine, and a context can be extended by some other context. Moreover, a machine can see one or several contexts.In one context, the set lists different sets of carriers, which define disjoint types in pairs. The key properties of constants are described by axioms. Since the theorem shows properties (to be proven) derived from already declared axioms, axioms can be established.

B. B - Notation

Few B notations which are mostly used in our system are presented here. A more elaborated justification of these can be found in [1], [11]. Lets assume that there are two sets named P & Q, then notation $\leftrightarrow$ expresses the set of relations between P and Q as

$$P \leftrightarrow Q = \mathbb{P}(P \times Q)$$

where $\times$ is known as the Cartesian product of set P and set Q. A mapping of element $p \in P$ and $q \in Q$ in a relation $R \in P \leftrightarrow Q$ is recorded as $p \mapsto q$. The domain of a relation $R \in P \leftrightarrow Q$ is the set of elements of P that R relates to some elements in Q expressed as

$$\text{dom(R)} = \{p \mid p \in P \ \wedge \exists \ q \ . \ (q \in Q \wedge p \mapsto q \in R)\}$$

Furthermore, the range of relation $R \in P \mapsto Q$ is defined as set of elements in Q related to some element in P :

$$\text{ran(R)} = \{ \ q \mid q \in Q \wedge \exists \ p \ . \ (p \in P \wedge p \mapsto q \in R \ ) \ \}$$

A function is a relation with some constraints. A function can be of two types: partial function ($\nrightarrow$) or a total function ($\rightarrow$). A partial function from set P to Q ($P \nrightarrow Q$) is a relation which relates an element in P to at most one element in Q.

A total function from set P to Q ($P \rightarrow Q$) is a partial function where dom(f)=P, i.e., each element of set P is related to exactly one element of set Q. Given $f \in P \nrightarrow Q$ and $p \in$ dom(f), f(p) represents the unique value that p is mapped to by f.

C. Rodin

In this paper the Rodin [3], [5] tool is used to formally verify the Event - B model of coordinated checkpoint process in distributed system. Rodin (Rigorous Open Development Environment for Complex Systems) is an extension of Eclipse IDE (Java based). It's an open-source, extensible tool for event - B specification and validation. The simplicity of use and configurability of the Rodin tool are its two important characteristics. The focus of the tool is on modeling. Models can be easily modified and model variations can be evaluated. The tool can also be extended easily.

III. System Model

In this section, an informal model of a distributed system is provided. The distributed system regarded in this paper is characterized by the following:

1) There is no shared memory between the processes and they send message to one another for communication through channels.
2) There is no loss of messages by channels and this is ensured by any end-to-end transmission protocol that makes the channels lossless(virtually) and first-in-first-out in delivery of messages.
3) Processes can malfunction, and when they do, all other processes must be informed of the failure within a certain amount of time.

The system model proposed here comprises the collection of sites on which the set of processes are running, they coordinate their checkpoints so that the subsequent global state is consistent. To allocate the timestamp to the communicating sites and the messages concerned, Lamport's logical clock is used. Proposed model saves two types of checkpoints on stable storage which are provided below:

- Permanent
- Tentative

It is not possible to undo a permanent checkpoint. It ensures that the computation necessary to enter the checkpoint state is not performed again. Nevertheless, it is possible to undone or alter a provisional checkpoint to be a permanent checkpoint.

Next, it is assumed that the algorithm is invoked by a single process to take a permanent checkpoint. Furthermore, it is also assumed that no site will fail while the algorithm is being executed. The model sends messages across lossless(virtually) and FIFO channels . The checkpoint algorithm is based on two phase commit protocol. The initiator q takes a tentative checkpoint in the first phase and asks to all processes to take tentative checkpoints. To assume the initiator's position of initiation and decision making, a hypothetical process called *daemon* is created. When q learns that all processes have taken provisional checkpoints, q chooses to make all provisional checkpoints permanent; otherwise, q decides to discard the provisional checkpoints. In the second phase, q's choice is spreaded and completed by all processes. The most recent set of checkpoints is always consistent as all or none of the systems take permanent checkpoints. This checkpoint decision is made in the same way that a request to take a provisional checkpoint is made. After taking a new permanent checkpoint, a process discards its old checkpoint [19].

IV. Event-B Model of Checkpoint Process in Distributed System

In the proposed model, a process known as the *daemon* process initiates the checkpoint process. This *daemon* broadcasts to all other participating processes (checkpoint Cohorts) a timestamped request message. checkpoint cohorts processes updates its local checkpoint number after receiving the request message and sends the *daemon* a time-stamped reply message. To allocate timestamp to the message, it increases its own local checkpoint number by one each time a message is sent by any process and the incremented value is assigned to the message as a timestamp. With the maximum timestamp value of the message sent and the latest checkpoint number at the time of message delivery, the received site updates its own local checkpoint number. The *daemon* site calculates the permanent checkpoint number after receiving the reply message from all checkpoint cohort processes. This global checkpoint number is shared with all stakeholders such that all processes can switch a temporary checkpoint into a permanent one.

V. Abstract Model of Checkpoint Process in Event - B

A. Context:

Context represents the static state of the system. Contexts may contain carrier sets, constants, and axioms. In the Context of the model we have declared two carrier sets. *PROCESS* and *PROCESS_MSG*. Carrier set *PROCESS* represents the set of processes running in the distributed system, and set *PROCESS_MSG* are the set of messages exchanged between processes. Three enumerated sets are also defined: *state*, *category* and *checkpointstate*.

The set *state* represents the state of the *daemon*. *Daemon* can be in any one of the four states:

1) *awaiting:* *Daemon* process waiting to receive the checkpoint response message from all checkpoint cohorts.
2) *received_all_responses:* *Daemon* process received checkpointing responses from all cohorts.
3) *permanent_ckpt_broadcast:* *Daemon* processes had broadcasted the permanent checkpoint number to all cohorts.
4) *idle:* *Daemon* process is idle.

The set *category* represents the category of the message communicated between *daemon* and the cohort processes. Message can be of any one category:

1) *tentative_ckpt_req:* This represents that the message type is tentative checkpoint request which is sent from *daemon* to checkpointing cohorts.
2) *tentative_ckpt_response:* This represents that the message type is tentative checkpoint response and is sent from checkpointing cohorts to *daemon*.
3) *permanent_ckpt_msg:* This represents that message is from *daemon* checkpoint cohorts and is permanent checkpoint message.

The set *checkpointstate* represents the checkpoint state of the processes. Any process can have any one checkpoint state at any given time:

1) *open:* represents that the checkpoint state is open.
2) *tentative:* represents that the checkpoint state of the process is tentative.
3) *permanent:* represents that the checkpoint state of the process is permanent.

MACHINE Checkpoint_Machine
SEES Checkpoint_Context
VARIABLES
daemon
daemon_status
sender
sent_msg
time_sent_msg
msg_category
deliver
time_response_msg
ckpt_state
no_of_responded_process
tentative_ckpt_no
permanent_ckpt_no
INVARIANTS
inv11: $tentative\_ckpt\_no \in PROCESS \rightarrow \mathbb{N}$
inv12: $permanent\_ckpt\_no \in \mathbb{N}$
inv1: $daemon \subseteq PROCESS$
inv2: $daemon\_status \in daemon \rightarrow state$
inv4: $sent\_msg \subseteq PROCESS\_MSG$
inv5: $time\_sent\_msg \in PROCESS\_MSG \rightarrow \mathbb{N}$
inv6: $msg\_category \in sent\_msg \rightarrow category$
inv7: $deliver \in PROCESS \leftrightarrow PROCESS\_MSG$
inv8: $time\_response\_msg \subseteq \mathbb{N}$
inv9: $ckpt\_state \in PROCESS \rightarrow checkpointstate$
inv10: $no\_of\_responded\_process \in \mathbb{N}$
inv3: $sender \in PROCESS\_MSG \nrightarrow PROCESS$
EVENTS
Initialisation
begin
act1: $ckpt\_state := \varnothing$
act2: $daemon\_status := \varnothing$
act3: $deliver := \varnothing$
act4: $sender := \varnothing$
act5: $sent\_msg := \varnothing$
act6: $time\_sent\_msg := \varnothing$
act7: $msg\_category := \varnothing$
act8: $time\_response\_msg := \varnothing$
act9: $no\_of\_responded\_process := 0$
act10: $tentative\_ckpt\_no := \varnothing$
act11: $permanent\_ckpt\_no := 0$
act12: $daemon := \varnothing$
end

Figure 1. Abstract model of checkpoint process in Event - B

B. Machine:

Macine represents the dynamic state of the system. Machines may contain variables, invariants, events and variants.

1) Variables:: In the machine of the model various variables have been defined, which are described below:

- *sender:* The variable sender represents the sender of message. The variable *sender* is a partial function from the set *PROCESS_MSG* to *PROCESS*. This is given in Fig.1

  inv 3: sender ∈ PROCESS_MSG ⇸ PROCESS

  A representation of the form $mm \mapsto pp \in sender$, represents that message *mm* was sent by process *pp*.
- *daemon:* daemon represents the checkpoint initiator process. *daemon* is any one process from the set *PROCESS*. This is defined in Invariant 1.

inv 1: daemon $\subseteq$ PROCESS

- *daemon_status:* Represents the status of the daemon (initiator process). Any process can be a daemon. It is a Total function from *daemon* to set *state*. It is defined in Invariant 2.

  inv 2: daemon_status $\in$ daemon $\rightarrow$ state

  Daemon status can be any one of the four values:
  - *awaiting*
  - *received_all_responses*
  - *permanent_ckpt_broadcast*
  - *idle*

  A representation of the form *daemon_status* $\in$ *daemon* $\mapsto$ *awaiting* represents that *daemon* is waiting to receive the response from checkpoint cohorts and did not received the response from all the cohorts.
- *sent_msg:* Represents the messages which was sent by the process.
- *time_sent_msg:* Represents the timestamp of the sent messages.
- *msg_category:* Represents the category of the sent message. Sent messages can of three types:
  - *tentative_ckpt_req*
  - *tentative_ckpt_response*
  - *permanent_ckpt_msg*

  This is given in Fig.1

  inv6 : msg_category $\in$sent_msg $\rightarrow$ category

  A representation of the form *msg_category* $\in$ *mm* $\mapsto$ *tentative_ckpt_req* represents that the category of the message *mm* is tentative checkpoint request.
- *deliver: deliver* represents the delivery of the message to a process. It is a Relation between *PROCESS* and *PROCESS_MSG*. It is defined in Fig.1

  inv7 : deliver $\in$ PROCESS $\leftrightarrow$ PROCESS_MSG

  A representation of the form *deliver* $\in$ *pp* $\mapsto$ *mm* represents that message *mm* has been delivered to process *pp*.
- *time_response_msg:* represents the timestamp of all reply messages to the coordinator.
- *ckpt_state:* represents the checkpoint state of each process. ckpt_state can be any one of the three values.
  - *open*
  - *tentative*
  - *permanent*

  ckpt_state is a Total function from set *PROCESS* to set *checkpointstate*. It is given in Fig.1

  inv9 : ckpt_state $\in$ PROCESS $\rightarrow$ checkpointstate

  A representation of the form *pp* $\mapsto$ *tentative* represents that process *pp* has taken *tentative* checkpoint.
- *no_of_responded_process:* represents the number of process responded for the request message for checkpoint creation. It is a set of Natural numbers.
- *tentative_ckpt_no:* represents tentative checkpoint number of each process. It is a Total function from *PROCESS* to Natural number. it is defined in the Fig.1.

  inv 11: tentative_ckpt_no $\in$ PROCESS $\rightarrow \mathbb{N}$

  A representation of the form *pp* $\mapsto$ *1* represents that process *pp* has taken a Tentatine checkpoint number *1*. *tentative_ckpt_no* store all the events which happens on the process and used for recovery purpose for that process.
- *permanent_ckpt_no:* Represents the permanent checkpoint number.

2) Events: : Various events have been defined in the machine. Informal information about the events are given below:

- Broadcasting checkpoint request message to the cohorts: In the event *Daemon_checkpoint_request_broadcast* in Fig.2 daemon process *pp* broadcast a checkpoint request message *mm* to all the cohorts to take the tentative checkpoint. The *grd4:* ensures that daemon always send

Event Daemon_ckpt_request_Broadcast ⟨ordinary⟩ $\widehat{=}$

any

pp

mm

where

grd1: $pp \in daemon$

grd2: $mm \in PROCESS\_MSG$

grd3: $daemon\_status(pp) = idle$

grd4: $mm \notin dom(sender)$

then

act1: $tentative\_ckpt\_no(pp) := tentative\_ckpt\_no(pp) + 1$

act2: $time\_sent\_msg(mm) := tentative\_ckpt\_no(pp)$

act3: $sender := sender \cup \{mm \mapsto pp\}$

act4: $daemon\_status(pp) := awaiting$

act5: $sent\_msg := sent\_msg \cup \{mm\}$

act6: $msg\_category(mm) := tentative\_ckpt\_req$

end

Figure 2. Broadcast operation of checkpoint process model

a fresh checkpoint request message to all the cohorts. *grd3:* ensures that the status of the *daemon* should be idle to sent a checkpoint request message. If the given guards are true then, timestamp is assigned to the checkpoint request message by incrementing the tentative checkpoint number of daemon process by 1, as given in Fig.2 (*act1: & act2:*). The status of the *daemon* is set to *awaiting* and category of the request message is set to *tentative_ckpt_req*.

– Checkpoint Request Message Receive by Checkpoint Cohort: In the event *Cohort_Checkpoint_Request_Receive* in Fig.3 all the checkpointing cohorts receive the checkpoint request broadcast and then updates its tentative checkpoint number with timestamp of the request message or its current timestamp value, whichever is higher. In fig.3

Event Cohort_Ckpt_Request_Receive ⟨ordinary⟩ ≙
any
pp
mm
dp
where
grd1: $pp \notin daemon$
grd7: $dp \in daemon$
grd8: $mm \mapsto dp \in sender$
grd9: $pp \mapsto mm \notin deliver$
grd2: $mm \in sent\_msg$
grd3: $msg\_category(mm) = tentative\_ckpt\_req$
grd4: $mm \notin deliver[\{pp\}]$
grd10: $mm \notin ran(deliver)$
grd5: $finite(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$
grd6: $ckpt\_state(pp) = open$
then
act1: $deliver := deliver \cup \{pp \mapsto mm\}$
act2: $tentative\_ckpt\_no(pp) := max(\{time\_sent\_msg(mm), tentative\_ckpt\_no(pp) + 1\})$
end

Figure 3. Checkpoint request receive operation of cohorts

grd4: and *grd10:* ensures that checkpoint request message *mm* is not delivered to the process *pp*, this is a fresh request message. *grd8:* ensures that sender of the checkpoint request message is *daemon*. *grd6:* ensures that state of receiver process of the checkpoint message should be *open*. If all the given guards are true then message *mm* is delivered to process *pp* in *act1 :* . In *act2 :* process *pp* takes tentative checkpoint number as maximum value of timestamp of the message *mm* or the tentative checkpoint number of the process incremented by 1.

– Checkpoint request Response by Cohort : In the event *Checkpoint_Cohort_Response* given in Fig.4 Every cohort process sends a timestamped response message to daemon. To allocate a timestamp to a response message, the process increases its tentative checkpoint number by one and uses that value as a timestamp. In *grd5:* it is ensured that request message must be delivered to cohort. The response message *mm* must be a fresh response message , it is ensured by *grd3:*. To send the response message to *daemon* , the checkpoint state of the

Event Ckpt_Cohort_Response ⟨ordinary⟩ ≙
any
pp
mm
m
where
grd1: $pp \notin daemon$
grd5: $pp \mapsto m \in deliver$
grd3: $mm \notin dom(sender)$
grd4: $ckpt\_state(pp) = open$
grd6: $msg\_category(m) = tentative\_ckpt\_req$
then
act1: $ckpt\_state(pp) := tentative$
act2: $sent\_msg := sent\_msg \cup \{mm\}$
act3: $msg\_category(mm) := tentative\_ckpt\_response$
act4: $sender := sender \cup \{mm \mapsto pp\}$
act5: $time\_sent\_msg(mm) := tentative\_ckpt\_no(pp) + 1$
act6: $tentative\_ckpt\_no(pp) := tentative\_ckpt\_no(pp) + 1$
end

Figure 4. Checkpoint response send operation of cohorts

cohort must be open. If all the guards are true

Event Cohort_Response_Submission_at_Daemon ⟨ordinary⟩ ≙
any
pp
mm
where
grd1: $pp \in daemon$
grd2: $mm \in sent\_msg$
grd3: $mm \notin deliver[\{pp\}]$
grd4: $msg\_category(mm) = tentative\_ckpt\_response$
grd5: $daemon\_status(pp) = awaiting$
then
act1: $deliver := deliver \cup \{pp \mapsto mm\}$
act2: $no\_of\_responded\_process := no\_of\_responded\_process + 1$
act3: $time\_response\_msg := time\_response\_msg \cup \{time\_sent\_msg(mm)\}$
end

Figure 5. Checkpoint response send operation of cohorts

then the checkpoint state of the process*pp* is set to *tentative* and message *mm* is added to the set *sent_msg*. Message category of the *mm* is set to *tentative_ckpt_response*, which represents that category is tentative checkpoint response (*act3:*). Timestamp of the response message *mm* is set to tentative checkpoint number incremented by value 1 (*act5:*).

– Cohort Response Submission at Daemon: In the event *Cohort_Response_submission_At_Daemon*

response messages from all the cohorts is delivered to the daemon process. It is given in Fig.5.

Whenever a daemon process receive a message from a cohort with category tentative_ckpt_response, it updates the no_of_responded_sites with 1. The value of the timestamp of the response message is also stored for the purpose of Permanent checkpoint computation. *grd3:* ensures that the response message *mm* is a fresh response message, and it is not already delivered to process *pp*. Message category of the response message *mm* should be *tentative_ckpt_response*, it is defined in *grd4:*. Status of the daemon should be *awaiting*.

If all the given guards are true then the response message *mm* should be deliverd to the *daemon* in *(act1:)*. Whenever receives the response from any cohort *daemon* the updates the *no_of_responded_process*, it is defined in (*act2:*). In (*act3:*) timestamp of the response message is added to the pool of all timestamp of the response messages.

– Permanent Checkpoint Computation: In the event *permanent_ckpt_computation daemon* must ensure that it has received the response messages from the all the cohorts(*grd2:*), it is given in Fig.6. *daemon* must ensure that its

Event permanent_ckpt_computation ⟨ordinary⟩ $\widehat{=}$
any
pp
where
grd1: $pp \in daemon$
grd2: $no\_of\_responded\_process = card(PROCESS) - 1$
grd3: $daemon\_status(pp) = awaiting$
then
act1: $daemon\_status(pp) := received\_all\_responses$
act2: $permanent\_ckpt\_no := max(time\_response\_msg)$
end

Figure 6. Permanent Checkpoint Computation by daemon

status should be *awaiting* before computing the permanent checkpoint number, it is defined in (*grd3 :*).

When all the given guards are true then the *daemon* changes its state from *awaiting* to *received_all_responses* (*act1:*), and maximum timestamp value from the received response messages is assigned to the *permanent_ckpt_no* (*act2:*).

– Broadcast of Permanent Checkpoint Number: In the event *Broadcast_Permanent_Ckpt_No* permanent checkpoint number, is broadcasted by *daemon* to all cohorts. It is given in Fig.7. *grd2:* ensures that message *mm* is a fresh per-

Event Broadcast_Permanent_Ckpt_No ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $pp \in daemon$
grd2: $mm \notin dom(sender)$
grd5: $time\_response\_msg \neq \varnothing$
grd3: $permanent\_ckpt\_no = max(time\_response\_msg)$
grd6: $daemon\_status(pp) = received\_all\_responses$
then
act1: $sender := sender \cup \{mm \mapsto pp\}$
act2: $sent\_msg := sent\_msg \cup \{mm\}$
act3: $msg\_category(mm) := permanent\_ckpt\_msg$
act4: $time\_sent\_msg(mm) := permanent\_ckpt\_no$
act5: $daemon\_status(pp) := permanent\_ckpt\_broadcast$
end

Figure 7. Broadcast operation of permanent checkpoint number

manent checkpoint number message. We put the *grd3:* to ensure that the value of permanent checkpoint number should be maximum of the timestamp of received messages. *grd6:* to ensure that status of the *daemon* should be *received_all_responses*. When all the given guards are true, message *mm* is added to the *sent_msg* in (*act2:*). Category of the sent permanent checkpoint message *mm* is set to the *permanent_ckpt_msg* (*act3:*). The timestamp of the message *mm* is set to the value of *permanent_ckpt_no* (*act4 :*). The status of the *daemon* is set to *permanent_ckpt_broadcast* (*act5:*).

– Permanent Checkpoint Message receive by Cohort: in the event *Cohort_Permanent_Ckpt_Message_Receive* when cohort process receives the permanent checkpoint number message from the *daemon* , it updates its tentative checkpoint number with the received permanent checkpoint message timestamp, it is given in Fig.8.

In *grd4:* of the Fig.8 it is checked that the category of the received permanent checkpoint message *mm* must be *permanent_ckpt_msg*. In the *grd6:* it is ensured that message *mm* should be a fresh permanent checkpoint message, it is not the duplicate message already received by the cohort . It is also ensured that the checkpoint state of the cohort must be *tentative* in *grd7:*.

If all the guards are true then message *mm* is

Event Cohort_permanent_Ckpt_Message_Receive ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $pp \notin daemon$
grd2: $mm \in sent\_msg$
grd4: $msg\_category(mm) = permanent\_ckpt\_msg$
grd5: $mm \in dom(sender)$
grd6: $pp \mapsto mm \notin deliver$
grd7: $ckpt\_state(pp) = tentative$
then
act1: $deliver := deliver \cup \{pp \mapsto mm\}$
act2: $tentative\_ckpt\_no(pp) := time\_sent\_msg(mm)$
end

Figure 8. Permanent checkpoint message receive operation by cohort

delivered to cohort *pp* and tentative checkpoint number of cohort *pp* is set to the timestamp of the received permanent checkpoint number message *mm*.

– Swithcing The Checkpoint State From Tentative to Permanent: In Fig.9 the event *Switching_from_Tentative_to_Permanent_State* cohort's checkpointing state is changed from *tentative* to *permanent*. Before switching the checkpoint state from *tentative* to *permanent* it is ensured that received permanent chackpoint message *mm* should have the category *permanent_ckpt_msg* and it is also delivered to the cohort process *pp*, it is defined in (*grd3:*) and (*grd7:*). The checkpoint state of the cohort process must be *tentative*.

Event Switching_from_Tentative_to_Permanent_state ⟨ordinary⟩ $\widehat{=}$
any
pp
mm
where
grd1: $pp \notin daemon$
grd2: $mm \in sent\_msg$
grd3: $msg\_category(mm) = permanent\_ckpt\_msg$
grd4: $ckpt\_state(pp) = tentative$
grd5: $pp \mapsto mm \in deliver$
then
act1: $ckpt\_state(pp) := permanent$
end

Figure 9. switching from tentative to permanent operation by cohort

When the given guards are true then the checkpoint state of the cohort process *pp* is changed to *permanent* to *tentative*.

VI. Conclusion

The checkpoint process presented in this paper allows any site to be the coordinator process, which we named as *daemon*. Our checkpoint algorithm is based on the concept of 2 phase commit protocol. If all the cohort processes are ready to take a permanent checkpoint then only all the tentative checkpoints are made permanent. This algorithm always having the consistent set of checkpoints.

Event - B is used to build the proposed model of the distributed system's checkpoint mechanism in this paper. Event - B supports the Incremental model development approach, and in each increment of the model we can refine our model. The Rodin tool is used to do the model checking and theorem proving. Rodin tool produces the proof obligations and check the consistency of the model during each refinement. Proof obligations can be discharged using the interactive and automatic prover of the Rodin. Around 70% oof the proofs are discharged by the automatic prover. Only 30% of the proofs requires interaction.

References

[1] J. Abrial. The b-book: assigning programs to meanings cambridge university press, 1996.

[2] J.-R. Abrial. Extending b without changing it (for developing distributed systems). In *1st Conference on the B method*, volume 11, 1996.

[3] J.-R. Abrial. A system development process with event-b and the rodin platform. In *International Conference on Formal Engineering Methods*, pages 1–3. Springer, 2007.

[4] J.-R. Abrial and J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[5] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[6] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for event-b. In *International Conference on Formal Engineering Methods*, pages 588–605. Springer, 2006.

[7] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal aspects of computing*, 14(3):215–227, 2003.

[8] D. Y. Bal Krishna Saraswat, Raghuraj Suryavanshi. A comparative study of checkpointing algorithms for distributed systems. *International Journal of Pure and Applied Mathematics*, 118:1595–1603, 2018.

[9] R. Banach. Retrenchment for event-b: Usecase-wise development and rodin integration. *Formal Aspects of Computing*, 23(1):113–131, 2011.

[10] D. Basin, A. Fürst, T. S. Hoang, K. Miyazaki, and N. Sato. Abstract data types in event-b-an application of generic instantiation. *arXiv preprint arXiv:1210.7283*, 2012.

[11] J.-L. Boulanger. *Formal methods applied to complex systems: implementation of the B method*. John Wiley & Sons, 2014.

[12] M. Butler and D. Yadav. An incremental development of the mondex system in event-b. *Formal Aspects of Computing*, 20(1):61–77, 2008.

[13] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[14] Clearsy. B4free tool homepage. www.b4free.com.

[15] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[16] S. Hallerstede. Justifications for the event-b modelling notation. In *International Conference of B Users*, pages 49–63. Springer, 2007.

[17] M. Jandl, A. Szep, R. Smeikal, and K. M. Göschka. Increasing availability by sacrificing data integrity-a problem statement. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 291c–291c. IEEE, 2005.

[18] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, 2000.

[19] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

[20] D. Manivannan, R. H. B. Netzer, and M. Singhal. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):623–627, 1997.

[21] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, 1999.

[22] D. Manivannan and M. Singhal. Asynchronous recovery without using vector timestamps. *Journal of Parallel and Distributed Computing*, 62(12):1695–1728, 2002.

[23] C. Métayer, J. Abrial, and L. Voisin. Event-b language, rodin deliverable d7. eu-project rodin (ist-511599)(2005), 2005.

[24] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[25] B. Randell. System structure for software fault tolerance. *Ieee transactions on software engineering*, (2):220–232, 1975.

[26] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, (2):183–194, 1980.