5

The Formalization and Analysis of CCR Protocol and Service

Bairong Zhang Alistair Munro Michael Barton

Centre for Communications Research, University of Bristol,
Bristol BS8 1TR, United Kingdom

Commitment, Concurrency and Recovery protocol and service defined by ISO are studied by representing them in the notation of CCS. The behaviour of the CCR protocol and service is investigated on the Concurrency Workbench, and desirable properties of CCR are expressed and analyzed in the modal mu-calculus. The formulae are checked on the Concurrency Workbench.

Keywords: B.4.5 Reliability, Testing and Fault-Tolerance C.2.2 Network Protocols C.2.4 Distributed Systems

1. INTRODUCTION

Commitment, Concurrency and Recovery (CCR) is an application-service-element standardised by ISO as an Open System Interconnection (OSI) application layer service. It is intended to be used by other ASEs for providing the functionality of two-phase commitment to application processes. It can be referenced whenever the processing of two or more application-entity invocations (or processes) in a distributed application needs to be organized as an atomic action, or when recovery is required after failure. CCR services are used intensively by ISO OSI standards such as Transaction Processing, and Job Transfer and Manipulation which supports the creation, distribution, and management of tasks for processing by computer systems in an open network. In either case, an activity can be performed as a single, multi-branch, multi-association atomic action, or as a sequence of atomic actions. However, the different modes of operation allow a wide range of configurations to be used by a distributed application and raise questions about the security and the complexity of the recovery process after failures. Hence, proving the integrity of CCR is considered a major challenge for formal description techniques[12].

The Calculus of Communicating Systems (CCS) [9] provides a semantic basis for reasoning about transition systems [8, 14, 1, 10]. Modal logic can be interpreted on labelled transition systems and it naturally describes the transitional behaviour of CCS processes. The modal mu-calculus is a modal logic extended by fixpoint operators; this is appropriate for describing temporal properties of transition systems. The Concurrency WorkBench (CWB) is a prototype automated system which caters for the manipulation and analysis of transition systems. In this paper, the CCR protocol and service are formalized in the notation of CCS, and *atomicity* and *liveness* properties of CCR are analyzed and expressed in the modal mu-calculus, and checked on the Concurrency Workbench.

D. Hogrefe et al. (eds.), *Formal Description Techniques VII*

2. CCR PROTOCOL AND SERVICE

2.1. Introduction

In an Open Systems Interconnection (OSI) environment, application processes may wish to communicate with each other for a wide variety of reasons. However, any communication requires certain common services independent of the specific application. CCR provides such a "common application service element" for a single association. It consists of a set of primitive services that can be used for starting and ending a specific sequence of distributed application operations despite application or communication failure. Thus the use of CCR allows a referencing specification to define its activity as an atomic action.

An atomic action is a specific set of related distributed application operations that may be characterized by the following properties: Atomicity – a set of related operations are either all performed or none of them are performed (exactly-once semantics); Consistency – the "real effects" of a set of related operations are performed accurately, correctly and with validity, with respect to application semantics; Isolation – the partial results of a set of related operations are not accessible, except by operations of the set; Durability – all the effects of the operations are not altered by any sort of failure.

Although the CCR discussed here is specifically the ISO version, it is linked very closely to distributed two-phase commitment protocols that have been used for many years by the distributed database and transaction processing communities.

2.2. CCR Service and Procedures

There are nine CCR services defined in [4, 5], a C-CANCEL service is added in [2, 3]. Table 1 lists the CCR services, the type of service, and the requestor of the service. Those CCR services form the following procedures: Initialization; Begin branch; Prepare; Offer commitment; Order commitment; Rollback; One-phase commitment; Read-only; Branch recovery; Order commitment and begin new branch; Rollback and begin new branch.

Table 1
CCR service

Service	Type	Requestor
C-INITIALIZE	Confirmed	Association-initiator
C-BEGIN	Optionally confirmed	Either CCR service-user
C-PREPARE	Non-confirmed	Branch-initiator
C-READY	Non-confirmed	Either CCR service-user
C-COMMIT	Confirmed	Either CCR service-user
C-ROLLBACK	Confirmed	Either CCR service-user
C-ONE-PHASE	Confirmed	Either CCR service-user
C-READ-ONLY	Non-confirmed	Either CCR service-user
C-RECOVER	Confirmed or optionally confirmed	Either CCR service-user
C-CANCEL	Non-confirmed	Either CCR service-user

The combined CCR procedures have a number of possible paths in which each path is composed of some of the above procedures starting with *Initialization* and then *begin*

branch. All the operations making up an atomic action are either completed successfully (leading to *commitment*) such that the *bound data* associated with an atomic action go to its *final state*; or they are *rolled back* such that the *bound data* is returned to its *initial state*; or when an atomic action is interrupted by an application or communication failure the system is left in a consistent state and *recovery* may take place at some later time.

3. CCS CONCEPT AND TOOLS

3.1. The Calculus of Communicating Systems

CCS [9] introduces five basic ways of constructing expressions for agents as the basic language. Agents are given a structured operational semantic as a labelled transition system, with labels drawn from a set of $Act = \mathcal{L} \cup \{\tau\}$ which range over by $\alpha, \beta \cdots$. Where labels $\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$. $\bar{\mathcal{A}}$ is a set of complements of labels in $\mathcal{A}$, where $\bar{\mathcal{A}} = \{\bar{l} | l \in \mathcal{A}\}$. τ is a distinguished silent action which is invisible to the observer; all other actions are visible. The set of agent expressions $\mathcal{E}$ range over by $E, F, \cdots$ has the following basic forms [14]:

- $\alpha.E$, a *Prefix* ($\alpha \in Act$). $\alpha.E$ is an agent which performs an action α and then become an agent E.
- $\Sigma_{i \in I} E_i$, a *Summation* (I an indexing set). $\Sigma_{i \in I} E_i$ behaves like one of E_i; as soon as one performs its first action the others are discarded.
- $E_1 | E_2$, a *Composition*. $E_1 | E_2$ is an agent in which E_1 and E_2 may proceed independently but may also communicate by performing a pair of complementary actions.
- $E \backslash \alpha$, a *Restriction* ($\alpha \subseteq \mathcal{L}$). $E \backslash \alpha$ is an agent which behaves like E, except that action α and its complementary action $\bar{\alpha}$ are forbidden.
- $E[f]$, a *Relabelling* (f a relabelling function). $E[f]$ is an agent derived from agent E by relabelling its action using the relabelling function f, where if $f(l) = l'$ then $f(\bar{l}) = \bar{l'}$.

The inference rules for those operations give the transition semantics. They are:

$$Act \quad \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad Sum_j \quad \frac{E_j \xrightarrow{\alpha} E'_j}{\Sigma_{i \in I} E_i \xrightarrow{\alpha} E'_j} \ (j \in I)$$

$$Com_1 \quad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \qquad Com_2 \quad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

$$Com_3 \quad \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\bar{l}} F'}{E|F \xrightarrow{\tau} E'|F'} \qquad Res \quad \frac{E \xrightarrow{\alpha} E'}{E \backslash L \xrightarrow{\alpha} E' \backslash L} \ (\alpha, \bar{\alpha} \notin L)$$

$$Rel \quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]} \qquad Con \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \ (A \stackrel{\text{def}}{=} P)$$

The basic transition system is $(\mathcal{E}, \{\xrightarrow{\alpha} | \alpha \in Act\})$. The interpretation of $P \xrightarrow{\alpha} Q$ is that P may evolve into Q by performing the observed action α, whereas $P \xrightarrow{\tau} Q$ means that P may evolve autonomously into Q.

Distinguishing between two agents by any external observation of their behaviour is achieved by a notion of bisimulation which is a standard device for defining behavioural equivalence for process algebras.

P and Q are strongly bisimilar or strongly equivalent, written $P \sim Q$. In strong bisimulation, every α action of one agent must be matched by an α action of the other – even for τ actions. If we merely require that each τ action be matched by zero or more τ actions; this yields a weaker notion of bisimulation. P and Q are (weakly) bisimilar or observation-equivalent, written $P \approx Q$ (refer to [9]).

3.2. Modal and Temporal Logics

Modal and temporal logics provide a repository of useful properties [13]. Modal logics are interpreted on labelled transition systems $(\mathcal{E}, \{\xrightarrow{\alpha} | \alpha \in Act\})$, where $\mathcal{E}$ is the set of agents; Act the action set; and $\xrightarrow{\alpha}$ the α transition relation. Its formulae are:

$$A ::= true \mid \neg A \mid A \wedge B \mid [\alpha]A \tag{1}$$

A formula is either the constant true formula; or a negated formula $\neg A$; or a conjunction of formulas $A \wedge B$; or a modalized formula $[\alpha]A$ whose intended meaning is: A holds after every action α. For each formula A the agents having the property A are defined as $\|A\|$:

$$\|true\| = \mathcal{E}$$

$$\|\neg A\| = \mathcal{E} - \|A\|$$

$$\|A \wedge B\| = \|A\| \cap \|B\|$$

$$\|[\alpha]A\| = \{E \in \mathcal{E} \mid \forall E' \; if \; E \xrightarrow{\alpha} E' \; then \; E' \in \|A\|\}$$

So agent E has the property $[\alpha]A$ if every E' such that $E \xrightarrow{\alpha} E'$ for any α in Act has the property A. The dual of $[\alpha]A$ is $\langle\alpha\rangle A$ which expresses that for some E' and some $\alpha \in Act$ such that $E \xrightarrow{\alpha} E'$.

Consider a recursive modal equation: $Z = \langle\alpha\rangle Z$. It can be viewed as expressing various properties of transition systems. Each of these properties is determined by a solution to the equation. On the CCS $\xrightarrow{\alpha}$ transition system there are infinitely many solutions to the above equation. Especially important are the maximal solution, which is the union of all solutions, and the minimal solution, the intersection of all solutions.

To express temporal properties it is useful to add recursively defined formulae. This leads to the *modal mu-calculus* whose formulae are obtained by adding the the description to the formulae (1).

$$A ::= \cdots \mid Z \mid \nu Z.A \tag{2}$$

Where Z range over a family of propositional variables, and νZ is a fixpoint operator. The formula $\nu Z.A$ expresses the property given by the maximal solution to $Z = A$. An important derived operator is $\mu Z = \neg\nu Z.\neg A[Z := \neg Z]$, where $A[Z := \neg Z]$ is the result of substituting $\neg Z$ for each free occurrence of Z in A. The formula $\mu Z.A$ expresses the property given by the minimal solution to $Z = A$ (refer to [13]).

Maximal fixed point formulae express invariance properties which can be described as safety properties – that nothing bad ever happens. Minimal fixed point formulae express eventuality properties which can be described as liveness properties – that something good does eventually happen.

3.3. The Concurrency Workbench

The Concurrency Workbench (CWB)[11] is an automated tool designed to provide machine assistance in analysis of concurrent systems described in CCS. The CWB is useful both in understanding and in reasoning about transition systems. The main functionalities we used in CWB are:

- Environment commands for binding agent identifier; printing environment; input file; output environment or agent into a file.
- Agent Commands for constructing state space of an agent; counting state size or checking the possible deadlocks in an agent.
- Equivalences and preorders commands for exploring the behaviour of an agent; determining whether or not two agents are related by bisimulation and testing equivalence and preorders; minimizing an agent observation-equivalence.
- Model checking command for checking whether a given process satisfies a specification formulated in the modal mu-calculus.

4. CCR REPRESENTATIONS IN CCS

The CCS descriptions of CCR protocol and service are based on [4, 5, 2, 3] CCR Protocol Machine (CCRPM) state table with the constraints of CCR service-user rules, and on the dynamic behaviour part of LOTOS description of CCR [6, 7].

4.1. Introduction to the Formal Description

A CCR is modeled in Figure 1 which consists of a CCR user and a CCRPM. The CCR *request* and *response* service primitives are sent by the CCR user via CCRsu, while the CCR *indication* and *confirm* service primitives are sent by the CCRPM via CCRsp. The CCR Application Protocol Data Units (APDUs) are sent or received through CCRpeer to another CCRPM. A CCR atomic action branch is composed of two CCR users: a superior (branch requestor) and a subordinate (branch acceptor) showed in Figure 2 and Figure 3.

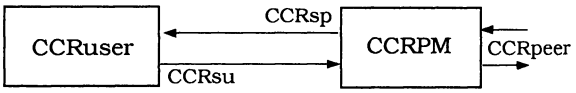

Figure 1. A CCR model

4.1.1. CCR Service

The CCR service [4, 2] CCRPM state table specifies the interrelationship between the current state of a CCRPM, the incoming service primitives and the resultant states of the CCRPM. An atomic action branch of CCR service is modeled as in Figure 2, where

sup CCRuser represents a superior, and *sub* CCRuser represents a subordinate. Issuing a *request* primitive from one CCRuser will result in receiving an *indication* primitive in another CCRuser, while issuing a *response* primitive from one CCRuser will result in receiving a *confirm* primitive in another CCRuser.

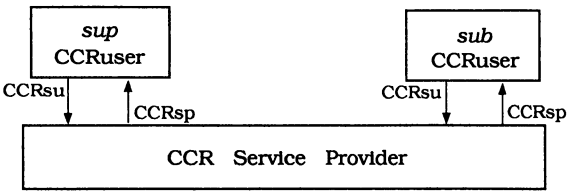

Figure 2. CCR service model

4.1.2. CCR Protocol

The CCR protocol [5, 3] CCRPM state table specifies the interrelationship between the current state of a CCRPM, the incoming events that occur, predicates and preconditions that determine what to do next, enablements that permit changes to data related to an atomic action, the action taken, outgoing events, and finally, the resultant state of the CCRPM. An atomic action branch of CCR protocol is modeled as in Figure 3. Receiving a *request* or *response* primitive from CCRsu in one CCRPM will result sending a RI (related to the Request and Indication primitives) or RC (related to the Response and Confirm primitives) CCR APDU to another CCRPM via CCRpeer, while receiving a RI or RC CCR APDU in one CCRPM from CCRpeer will result sending an *indication* or *confirm* primitive to its own CCRuser via CCRsp.

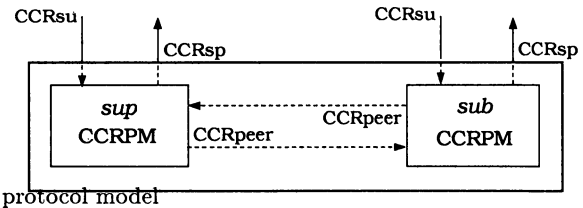

Figure 3. CCR protocol model

4.2. CCS Specification

In this paper we are concerned with a CCR atomic action branch which is composed of two CCR users: a CCR superior and a CCR subordinate. A complementary action of

l representation in CWB is $'l$.

4.2.1. Service Primitives Representation

The CCR service primitives C-INITIALIZE, C-BEGIN, C-PREPARE, C-READY, C-COMMIT, C-ROLLBACK, C-READ-ONLY, C-CANCEL, C-RECOVERY and C-ONE-PHASE are represented by **i**, **b**, **p**, **d**,**m**, **l**, **r** **c**, **rc** and **o** respectively. Four service primitives: request, indication, response and confirm are represented by **req**, **'req**, **rsp**, **'rsp** respectively. The recovery service primitive has five values which can be looked upon as five actions. They are **ready**, **commit**, **unknown**, **retry** and **done**.

4.2.2. CCR APDU Representation

There are sixteen CCR APDUs related to ten CCR service elements. Receiving and issuing those CCR APDUs via CCRpeer form the incoming and outgoing events in the CCRPM state table. Issuing CCR APDUs are represented as **tbga**, **tbgn**, **tcan**, **tcma**, **tcmt**, **tcmtbg**, **tina**, **tini**, **topa**, **toph**, **tprp**, **tro**, **trba**, **trbk**, **trbkbg**, **trca**, **trcv** and **trdy**. Receiving CCR APDUs are represented by adding $'$ in front of each issuing CCR APDU representation.

4.2.3. Global Constraints of the CCR

In the CCR service specification, issuing and receiving the service primitives from *sup* ccr_{user} and *sub* ccr_{user} are presented as a restricted composition in CCS. In the CCR protocol specification, issuing and receiving the CCR APDUs from *sup* ccr_{peer} and *sub* ccr_{peer} are presented as a restricted composition in CCS. The state change is caused by those incoming and outgoing events which form the dynamic behaviour of CCR.

An atomic action *Branch* is composed of a superior and a subordinate.

$$Branch = (SupCCEP|SubCCEP)\backslash\{outgoing\ event,\ incoming\ event\}$$

where CCEP means CCR Connection End Point.

4.2.4. Constraints of the Superior and the Subordinate

The CCR procedures in the superior side and subordinate side can be divided into a normal sequence and a recovery sequence. The normal sequence is made up of all the service primitives and related CCR APDUs, based on the CCRPM state table. The recovery sequence is made up of the C-RECOVERY service primitive and five actions, based on the Branch Recovery Machine (BRM) state table. The normal sequence and the recovery sequence are represented as a summation in CCS.

The CCS specification of CCR presented and studied here has five versions. The first one is based on the dynamic behaviour part of the LOTOS description of CCR [6] (see Appendix A, B, C). Only six basic CCR service C-BEGIN, C-PREPARE, C-READY, C-COMMIT, C-ROLLBACK and C-RECOVERY are concerned. This specification represented as $Branch_l$. Others are constructed from [4, 5, 2, 3] the CCRPM state table and the BRM state table. The constraints of CCR service-user rules are applied in forming a restricted composition on CCS. They are represented as $Branch_{s.91}$, $Branch_{p.91}$, $Branch_{s.94}$, $Branch_{p.94}$, BRM_{91} and BRM_{94}.

In an atomic action branch, after *committed* or *rolled back*, both sides will exit from the branch. If C-BEGIN request primitive is issued with C-COMMIT or C-ROLLBACK request primitive, the superior and the subordinate will start a new atomic action branch.

5. REASONING ABOUT CCR

A property which should be satisfied by the procedures of CCR service and protocol is to be deadlock free. Deadlock free means being always capable of executing an action. It can be expressed as a maximal fixed point formula:

$$s \models \nu Z.(\langle . \rangle true \wedge [.]Z)$$

This states that at every point on every path from s, executing an action is always possible. This formula was checked on the Concurrency Workbench. A deadlock can be found by the command '**fd**' on the Concurrency Workbench. The results from the Concurrency Workbench show that: Agents $Branch_l$, $Branch_{p.91}$, $Branch_{s.94}$ and $Branch_{p.94}$ are deadlock free. There is a deadlock state $(L3|I)$ in $Branch_{s.91}$. This deadlock can be avoided by adding an incoming event '**breq** from the state $L3$, then change to the state $A2$. In the amendment [2, 3] the procedures of CCR service and protocol are deadlock free. This property guarantees that CCR will always respond in some way.

The states can be counted by the '**states**' command. The results show that state D1 and state H4 are unreachable in all the *Branch* specification. D1 is a state when a C-READY request primitive has been issued, and then a C-READY indication primitive is received. H4 is a state when a C-ROLLBACK-BEGIN request primitive has been issued, and then a C-ROLLBACK indication primitive is received. The C-ONE-PHASE service has no effect on the CCRPM state table in [2]. The states K1 and J in [2] are unreachable as they are both related to the C-ONE-PHASE service. The states checked from BRM_{91} show that states from BRM5 to BRM9 are all unreachable, but they are reachable states in BRM_{94}.

5.1. Bisimulation Agent

To analyze the behaviour of a *Branch* agent, we can construct and analyze a simpler agent Br by using the relation commands. Where

$$\begin{aligned} Br &= (A|B)\backslash\{a,\ b\} \\ A &= a.A + b.P \\ B &= {}'a.B + {}'b.P \\ P &= e.P \end{aligned}$$

The Br agent is constructed from two agents A and B by restricted composition. The agents A and B are relevant to the superior and the subordinate in *Branch* agent. After a communication or synchronization via a pair of complementary actions a or b, A and B can either go back to the branch or exit from the branch. The results from the Concurrency Workbench show that *Branch* and Br are observation equivalent and observation congruent. This implies that in an atomic action branch, after a communication, both sides may exit from the branch or begin a new atomic action branch.

5.2. Liveness

How should we interpret the term *live* for the procedures of CCR? In an atomic action branch, there are a number of choices for the superior and subordinate. In a normal sequence one choice is that after committed the superior and subordinate will both exit from the branch and release the bound data in the final state. Another choice is to force the completion of an atomic action, in this case the superior and subordinate rollback and release the bound data in the initial state. A further choice is a C-BEGIN request primitive issued with C-COMMIT or C-ROLLBACK request primitive: in this case the superior and subordinate will go back and start a new atomic action. While in a recovery sequence one choice is that after a **recovery committed** request, if **recovery done** is confirmed, the superior and subordinate will both exit from the branch and release the bound data in the final state. A second choice is that after a **recovery ready** request, if **recovery unknow** is confirmed, the superior and subordinate will both exit from the branch and release the bound data in the initial state. The final choice is that if **recovery retry** is confirmed, the superior and subordinate will both go to the states where the recovery sequence started.

As CCS is a rigorous semantics language, both the interpretations and assumptions may be formulated precisely. To interpret the term *live*, we say the procedure of CCR is live if whenever at some point in a computation, all those choices described above are possible choices, and will eventually happen after some action. Since all actions in the *Branch* are complementary actions and cannot be observed outside of the *Branch*, we add an extra action at the end of each possible choice. For those paths which will exit from the *Branch* we add $exitp_i$ and $exitb_i$ in the superior and subordinate respectively, and for those paths which will go back to *Branch* we add $backp_i$ and $backb_i$ in the superior and subordinate respectively, where i range over a finite integer set. This interpretation of *live* can be expressed in the following minimal fixed point formulae:

$$Branch \models \bigwedge_i \mu Z.(((\langle exitp_i \rangle true \wedge \langle exitb_i \rangle true) \vee \langle . \rangle Z)$$

$$Branch \models \bigwedge_i \mu Z.(((\langle backp_i \rangle true \wedge \langle backb_i \rangle true) \vee \langle . \rangle Z)$$

This means that there is some point along some path in *Branch* consisting of some action at which both $exitp_i$ and $exitb_i$ may be performed or both $backp_i$ and $backb_i$ may be performed. The formula:

$$Branch \models$$

$$\bigwedge_i \mu Z.((((\langle backp_i \rangle true \wedge \langle backb_i \rangle true) \vee (\langle exitp_i \rangle true \wedge \langle exitb_i \rangle true)) \vee [.]Z)$$

means that along every path, containing infinitely many visible actions, one of the possible choices described above will happen eventually.

5.3. Atomicity

To say that the procedure of CCR preserves atomicity means that a set of related operations are either all performed or none of them are performed. As we have described above, superior and subordinate have several executing path choices and related bound data will be released in different states accordingly. Thus to establish that the procedure of CCR preserves atomicity it suffices to show that the superior and the subordinate cannot evolve into a state in which the bound data is in *initial state* in one side while it is in *final state* in the other side; or one exits from the branch while the other one stays; or one is in a normal sequence while the other one is in a recovery sequence. That is, it is enough to show that $exitp_i$ and $exitb_j$ where i is different from j can not be performed simultaneously; and both $backp_i$ and $backb_j$ where i is different from j can not be performed simultaneously; and both $exitp_i$ and $backb_j$ can not be performed simultaneously; and both $backp_i$ and $exitb_j$ can not be performed simultaneously. If none of above actions can be performed, then superior and subordinate will go through the same sequence and the bound data will be released in a required state.

This property can be expressed in the following maximum fixed point formulae:

$$Branch \models \bigwedge_i \bigwedge_{j \neq i} \nu Z.(\neg(\langle exitp_i \rangle true \wedge \langle exitb_j \rangle true) \wedge [.]Z)$$

$$Branch \models \bigwedge_i \bigwedge_{j \neq i} \nu Z.(\neg(\langle backp_i \rangle true \wedge \langle backb_j \rangle true) \wedge [.]Z)$$

$$Branch \models \bigwedge_i \bigwedge_{j \neq i} \nu Z.(\neg(\langle exitp_i \rangle true \wedge \langle backb_j \rangle true) \wedge [.]Z)$$

$$Branch \models \bigwedge_i \bigwedge_{j \neq i} \nu Z.(\neg(\langle backp_i \rangle true \wedge \langle exitb_j \rangle true) \wedge [.]Z)$$

This means that at every point on every path from *Branch*, consisting of any of the actions at which both $exitp_i$ and $exitb_j$; $backp_i$ and $backb_j$; $exitp_i$ and $backb_j$; $backp_i$ and $exitb_j$ cannot be performed simultaneously is always true. In this case, that superior and subordinate release the bound data in the required state is always true; or superior and subordinate go to the normal sequence or recovery sequence simultaneously is always true; or superior and subordinate exit from the *Branch* simultaneously or restart a new *Branch* simultaneously is always true, hence the atomicity property is preserved.

6. CONCLUSION

In this paper the procedures of CCR service and protocol have been formalized in the notation of CCS, and desirable properties of CCR have been interpreted and expressed by the modal mu-calculus and checked on the Concurrency Workbench.

A complex, dynamic system can be viewed as being composed of several parts called agents. Although the decomposition of a system into agents depends upon the user's intent, not upon the entities, there are often common services required to coordinate individual activities within the distributed application. CCR, which provides guarantees of atomicity, is one such service. Although the protocol analysed in this paper is described

in terms related directly to the ISO OSI version, the analysis has concentrated on key properties that apply to distributed two-phase commitment in general.

From the implementation viewpoint the CCR protocol and service can be constructed as a finite state machine. Realising this directly in CCS, based on the automata given in the specification, requires a lot of states to be specified. This results in a system that is too large to be analyzed by the Concurrency Workbench. From the formal analysis viewpoint, we wish to know whether the procedures of CCR are workable in general, hence we can concentrate on those important parts which construct the main procedures of CCR and contribute to the main properties of CCR, and ignore some less important details, such as some actions related to each service primitive. We assume that actions which are related to each incoming event are performed correctly, and under the correct execution of the service primitives we are concerned whether the procedures of CCR service and protocol are workable. A more strict formal analysis of CCR, consisting of data structures in more detail is needed but this is restricted by the current tools available.

Future work will consider compositional issues: both where the protocol is used by a client application, thus capturing the semantics of an entire system; and, also, where the essence of the protocol is retained but optimised by incorporating its behaviour directly in the application protocol. In either case, the obligation is incurred to demonstrate that the system retains the properties demonstrated here.

7. Acknowledgments

The authors are grateful for the support of Prof.J.P.McGeehan of the Centre for Communications Research in this work. It was funded in part by the Science and Engineering Research Council of the UK under grant GR54399 and by the Commission of the European Union under Esprit Project 6441.

REFERENCES

1. Glenn Bruns, Stuart Anderson *The Formalization and Analysis of a Communications Protocol*, LFCS Report ECS-LFCS-91-137, University of Edinburgh, 1991
2. *Commitment, Concurrency and Recovery Service - Amendment 1: enhancements* February 1994
3. *Commitment, Concurrency and Recovery Protocol - Amendment 1: enhancements* February 1994
4. Jim Quigley *ISO 9804/AM1 – Working draft # 4*, ACSE Project, 1991
5. *Amendment to CCR protocol specification (ISO/IEC 9805) covering enhancements– Working draft 4*, September 1991
6. ISO/IEC JTC 1/SC 21 N 8502 *Revised Text of PDTR 11589, LOTOS Description of CCR Service* February 1994
7. ISO/IEC JTC 1/SC 21 N 8503 *Revised Text of PDTR 11590, LOTOS Description of CCR Protocol* February 1994
8. Joachim Parrow, *Verifying a CSMA/CD-protocol with CCS* Protocol Specification, Testing, and Verification VIII IFIP pp373-384, 1988
9. R. Milner, **Communication and Concurrency**, Prentice-Hall international, 1989

10. M. J. Morley *Modelling British Rail's Interlocking Logic: Geographic Data Correctness*, LFCS Report ECS-LFCS-91-186, University of Edinburgh, 1991
11. Faron Moller, *The Edinburgh Concurrency Workbench(Version 6.1)*, Department of Computer Science, University of Edinburgh, October, 1992
12. Alistair Munro, *Job Transfer and Manipulation(JTM)*, Centre for Communications Research, University of Bristol, UKUUG and UKnet Winter Technical Meeting, Dec. 1989
13. C. Stirling, *Temporal Logics for CCS*, Lecture Notes in Computer Science 354, pp 660-672, 1989
14. D. J. Walker *Automated Analysis of Mutual Exclusion Algorithms using CCS*, LFCS Report ECS-LFCS-89-91, University of Edinburgh, 1989

APPENDIX

A. Global constraints of the CCR service

$$Branch = (SupCCEP|SubCCEP)\backslash\{breq, brsp, preq, dreq, mreq, mrsp, lreq, lrsp, rcreq, rcrsp, mbreq, lbreq, ready, commit, unknown, retry, done\}$$

B. Constraints of the superior and subordinate

B.1. Normal sequence of the superior

$$SupCCEP = SupAction + SupRecovery$$

$$SupAction = breq.ContSupAction$$

$$ContSupAction = 'brsp.(preq.ContSupAction1 + ContSupAction1) + preq.('brsp.ContSupAction1 + ContSupAction1) + ContSupAction1$$

$$ContSupAction1 = 'dreq.SupDecision +' lreq.(lrsp.Process + lbreq.'lrsp.ContSupAction + lreq.'lrsp.Process) + lreq.SupRollSend1 + lbreq.SupRollSend2$$

$$SupDecision = mreq.'mrsp.Process + lreq.'lrsp.Process + mbreq.'mrsp.ContSupAction + lbreq.'lrsp.ContSupAction$$

$$SupRollSend1 = 'lrsp.Process +' lreq.lrsp.Process$$

$$SupRollSend2 = 'lrsp.ContSupAction +' lreq.lrsp.ContSupAction$$

B.2. Recovery sequence of the superior

$$SupRecovery = 'rcreq.'ready.SupRecResponse + rcreq.commit.SupRecActions$$

$$SupRecResponse = rcreq.commit.SupRecActions + rcrsp.(unknown.Process + retry.SupRecovery)$$

$$SupRecActions = 'rcrsp.('done.Process +' retry.SupRecovery)$$

B.3. Normal sequence of the subordinate

$$SubCCEP = SubAction + SubRecovery$$

$$SubAction = 'breq.ContSubAction$$

$$ContSubAction = brsp.('preq.ContSubAction1 + dreq.ContSubAction2 + Rollback) +' preq.(brsp.ContSubAction1 + dreq.ContSubAction3 + Rollback) + dreq.ContSubAction2 + Rollback$$

$$ContSubAction1 = dreq.ContSubAction3 + Rollback$$

$$ContSubAction2 = 'preq.ContSubAction3 + ContSubAction3$$

$$ContSubAction3 = 'mreq.mrsp.Process +' lreq.lrsp.Process + 'mbreq.mrsp.ContSubAction + 'lbreq.lrsp.ContSubAction$$

$$Rollback = 'lreq.(lrsp.Process + lreq.'lrsp.Process) + lreq.('lrsp.Process +' lreq.lrsp.Process + 'lbreq.lrsp.ContSubAction) + 'lbreq.(lrsp.ContSubAction + lreq.'lrsp.ContSubAction)$$

B.4. Recovery sequence of the subordinate

$$SubRecovery = rcreq.ready.SubRecResponse + 'rcreq.'commit.SubRecAction$$

$$SubRecResponse = 'rcrsp.('unknown.Process +' retry.SubRecovery) + 'rcreq.'commit.SubRecAction$$

$$SubRecAction = rcrsp.(done.Process + retry.SubRecovery)$$