Revisiting Snapshot Algorithms by Refinement-based Techniques

Manamiary Bruno Andriamiarina, Dominique Méry, Neeraj Kumar Singh

Revisiting Snapshot Algorithms by Refinement-based Techniques

Manamiary Bruno Andriamiarina
Université de Lorraine, LORIA
Vandœuvre-lès-Nancy, France
Email: manamiary.andriamiarina@loria.fr

Dominique Méry
Université de Lorraine, LORIA
Vandœuvre-lès-Nancy, France
Email: dominique.mery@loria.fr

Neeraj Kumar Singh
University of York
York, UK
Email: neeraj.singh@cs.york.ac.uk

***Abstract*—The snapshot problem addresses a collection of important algorithmic issues related to the distributed computations, which are used for debugging or recovering the distributed programs. Among the existing solutions, Chandy and Lamport propose a simple distributed algorithm. In this paper, we explore the correct-by-construction process to formalize the snapshot algorithms in distributed system. The formalization process is based on a modeling language Event B, which supports a refinement-based incremental development using RODIN platform. These refinement-based techniques help to derive a correct distributed algorithm. Moreover, we demonstrate how this class of other distributed algorithms can be revisited. A consequence is to provide a fully mechanized proof of the distributed algorithms.**

***Keywords*-Distributed algorithms; correctness by construction; snapshot; verification**

I. Introduction

The snapshot problem is a fundamental aspect of distributed computations and distributed applications, since it produces a global state of a distributed system at a particular instant. It is a photography of a global state made up of local states of each process and communication channels. Several solutions for the snapshot problem have been published, among them we consider the seminal algorithm of Chandy and Lamport [9, 21, 23]. The snapshot computation is motivated by several applications as, for instance, the verification of stable properties like deadlock, successful termination and debugging of the distributed program using *safe* configuration. Snapshot algorithms constitute a pertinent collection of case studies for evaluating strengths and weaknesses of formal techniques like model-checking [11, 12] and theorem prover [11, 19, 22]. The correct-by-construction paradigm [15] offers an alternative approach to prove distributed algorithms and to derive the *correct* distributed algorithms through the reconstruction of a target algorithm using stepwise refinement and validated methodological techniques [2, 5]. It appears that the refinement is a key concept for *organizing* the re-development of an existing distributed algorithm [2] to discover a new set of distributed algorithms [7] by reusing or replaying with the former development.

In this paper, we focus on the distributed snapshots for specific problems. The prime objective is to solve a problem using refinement techniques and to provide an evidence of correctness of given solutions, which are obtained through the *correct-by-construction* process. We are mainly interested by providing *recipes* for using the Event B framework and refinement for developing the distributed algorithms. Massingill and Chandy[16] introduce *archetypes* for facilitating parallel program design; more recently, Chandy et al [8] propose the refinement of formal archetypes to produce verified distributed software using the theorem prover PVS. The conceptual idea of the archetypes is very close to the design patterns in the software engineering domain. Refinement plays a central role in the integration of different archetypes and constitutes the semantical glue for ensuring the correctness of the resulting process. This approach is based on the use of PVS, which is employed to prove the properties of problems modelled using archetypes. Our *recipes* are conceptually close to the notion behind the archetypes and our aims are to use the Event B framework for developing correct-by-construction distributed algorithms, and enrich a collection of complex distributed algorithms (Project RIMEL: http://rimel.loria.fr). Another objective is to show the power of the *correct-by-construction* process and our *recipes* through the re-development and derivation of already existing and correct snapshot algorithms like the Chandy and Lamport algorithm [9], or the algorithm of Lai and Yang [13]. Finally, the snapshot problem is already considered as a case study for illustrating the strength of rewriting logic [18] and we think that our development may help a reader to understand the behavioral theory of snapshot algorithms.

The paper is organized as follows. Section 2 defines the snapshot problem in distributed systems. Section 3 introduces notations of Event B and the formal activities of a global system. Section 4 presents refinement-based development of the snapshot algorithm, where we describe the OBSERVATION model for stating *what* we have to compute. Section 5 introduces the computation of a snapshot in the PROCESS model, which simulates the OBSERVATION model. The global architecture of the refinement-based design is similar to the classical distributed algorithms [9, 13]. Section 5 concludes this paper along with the future work.

II. The Snapshot Problem

This section presents an abstract overview of the snapshot problem, which helps to understand our proposed solution. We consider a message passing model which formulates a distributed algorithm using a finite set of processes and channels. A direct channel connects each pair of nodes and a list of transformations is attached to each node, which performs either local actions or communications actions. The communication mechanism is supposed to be reliable, which guarantees that the channel does not lose any data packets.

For each node (process), a set of events (*send*, *receive* and *internal* events) is defined. A *partial ordering* called *local causal order* (denoted $<_p$ for a process (p)), induced by the local sequentiality of each process is defined. The following relationship $e_i <_p e_j$, between two events e_i and e_j of a process (p), indicates that e_i occurs before e_j. A cut C of a local set of events is a subset of events satisfying the relationship : $\forall p \in P, e, f \in C \cdot f \in L \wedge e <_p f \Rightarrow e \in C$. P is a set of processes and L is a set of *pre-shot* events (happening *before* the cut C).

Another *ordering* called *causal order* (denoted $<$) is defined as well. It is the smallest relation containing the *local causal orders* ($<_p$) and satisfying the *send/receive* ordering between processes. The relationship $e_m < e_n$, between two events e_m and e_n of a distributed system, means that e_m occurs before e_n :

1) If e_m and e_n are local to a process (p), then $e_m <_p e_n$.
2) If e_m represents the sending of a message, then e_n formulates the receiving of the message.
3) There exists another event e_k, such that $e_m < e_k$ and $e_k < e_n$.

A consistent cut C of a set of events of a distributed algorithm is a subset of events, which satisfies the following relationship : $\forall e, f \in C \cdot f \in L \wedge e < f \Rightarrow e \in C$.

A snapshot S is a global state of a distributed system, which is defined by a set of local states of nodes, and a set of channels states, produced by either internal actions or communication actions. The snapshot S is meaningful and feasible, if there exists an execution producing the global state, and a set of messages is successfully passed through each channel ($p \mapsto q$) of the distributed system, where a set of messages is sent by the node (p) and the sending messages are received by the node (q).

The following theorem [21] relates the notions of cut and snapshot :

Theorem 1 *A snapshot S induced by a cut C is meaningful if, and only if, C is consistent if, and only if, S is meaningful.*

The aim of the snapshot algorithm is to compute a global state of the system from the local states or equivalently a consistent cut. We investigate different steps for deriving two well-known snapshot algorithms [9, 13] using proof-assisted stepwise development.

III. Stepwise Design of Distributed Algorithms

The *correct-by-construction* paradigm promotes the development of algorithms using a progressive and incremental approach. The key concept is the refinement which provides linking between *discrete* models by preserving safety properties. The Event B modeling language designed by Abrial [1] borrows features from formal modeling languages like UNITY [10], TLA$^+$ [14], action systems [3, 4]; those modeling languages share common aspects and especially the refinement concepts. The Event B is supported by an open environment RODIN integrating formal features for developing discrete logico-mathematical models. The Event B provides structures for expressing the reactive systems as a set of actions called events and maintaining a list of assertions called (inductive) invariants. These invariants formulate safety properties. We express our design for modeling the distributed algorithms in the Event B using correct-by-construction approach, which is also our primary objective of this work. We recall basic concepts of the Event B modeling language [1] and a formal development tool called RODIN [20].

A. Modelling actions over states

The event-driven approach [1] is based on the B notation. It extends the methodological scope of basic concepts in order to take into account the idea of *formal models*. A formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*; an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the events. Here, we briefly recall definitions and principles of formal models and explain how they can be managed by tools [20].

Modifications over state variables are stated by events. An event has two main parts: a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalized substitution. An event can take one of the three normal forms described in figure 1 and is associated with a before-after predicate $BA(x, x')$, which describes the event as a logical predicate expressing the relationship between values of the state variables just before (x) and just after (x') the "execution" of the event (see Fig. 1).

Proof obligations (INV 1 and INV 2) are produced by the tool RODIN [20] from events in order to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before-after predicate, $BA(e)(x, x')$, of each event e (see Table 1). Note that it follows from the two guarded forms of the events and this obligation can be trivially discharged in case of false condition of the guard. When this is the case, the event is said to be *disabled*. The proof obligation FIS expresses the feasibility of the event e with respect to the invariant I.

Event e	*Before-After Predicate BA(e)(x, x')*
BEGIN *x :| P(x, x')* *END*	*P(x, x')*
WHEN *G(x)* *THEN* *x :| (Q(x, x'))* *END*	*G(x) ∧ Q(x, x')*
ANY t *WHERE* *G(t, x)* *THEN* *x :| (R(x, x', t))* *END*	*∃t. (G(t, x) ∧ R(x, x', t))*

	Proof obligations
(INV1)	*Init(x) ⇒ I(x)*
(INV2)	*I(x) ∧ BA(e)(x, x') ⇒ I(x')*
(FIS)	*I(x) ∧ grd(e)(x) ⇒ ∃y.BA(e)(x, y)*

Figure 1. Events and proof obligations

B. Describing the network and its activities

A network of processes is simply defined by a set of processes P, a set of channels between processes, namely C. We assume that M is a set of messages that can transit along channels. Each process may have a local state and a set of local states is $PStates$. The communication network is modelled by a structure called NETWORK. The network is supposed to be fixed (channels are not modified or created or deleted) and connected.

CONTEXT NETWORK
SETS
$P, M, PStates$
CONSTANTS
C ...
AXIOMS
$axm1 : C \subseteq (P \times P) \setminus id$
...
END

C. Describing the current system

The snapshot algorithm captures a set of actions modifying a set of variables, through the observation of the current distributed system. Hence, our modeling process states that the existing system simulates a new set of modifications in the current state. A model SYSTEM describes the general activities of the distributed system.

These activities are a) *internal* and b) *external* (interactions between nodes) operations modelled by the following events: Internal-local: to modify a local state of a process (p); Internal-message: to modify a local set of messages using a process (p); Sending: a process (p) sends a message to a process (q); Receiving: a process (q) receives a message from a process (p). After each operation, the time-stamp ($o(p)$) of a process (p) is incremented, and a trace of activities (either *internal/local* or *external*) is added to history ($h(p)$) of the process (p).

MACHINE SYSTEM
...
EVENT Internal-local ...
EVENT Internal-Message ...
EVENT Sending ...
EVENT Receiving ...
...
END

A new step expresses the observation of the current system by another process which is defined by a refinement of the current model. In the next section, we define the refinement and apply it for the observation.

IV. Incremental Proof-based Development

A. Model Refinement

The refinement of a formal model allows us to enrich a model in an *incremental* way which is the foundation of the *correct-by-construction* [15] approach. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model in a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines $skip$, (3) no new event takes control for ever, and (4) relative deadlock-freeness is preserved. Details of the formulation of these proofs follows.

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x, y)$. If $BA(e)(x, x')$ and $BA(f)(y, y')$ are respectively the abstract and concrete before-after predicates of the same event, respectively e and f, we have to prove the following statement, corresponding to proof obligation (1):

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(f)(y, y') \ \Rightarrow \ \exists x' \cdot (BA(e)(x, x') \ \wedge \ J(x', y'))$$

Now, proof obligation (2) states that $BA(f)(y, y')$ must refine $skip$ ($x' = x$), generating the following simple statement to prove (2):

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(f)(y, y') \ \Rightarrow \ J(x, y')$$

For the third proof obligation, we must formalize the notion of the system advancing in its execution; a standard technique is to introduce a variant $V(y)$ that is decreased by each new event (to guarantee that an abstract step may occur). This leads to the following simple statement to prove (3):

$$I(x) \ \wedge \ J(x, y) \ \wedge \ BA(f)(y, y') \ \Rightarrow \ V(y') < V(y)$$

Finally, to prove that the concrete model does not introduce additional deadlocks, we give formalisms for reasoning about the event guards in the concrete and abstract models: $\mathsf{grds}(AM)$ represents the disjunction of the guards of the events of the abstract model, and $\mathsf{grds}(CM)$ represents the

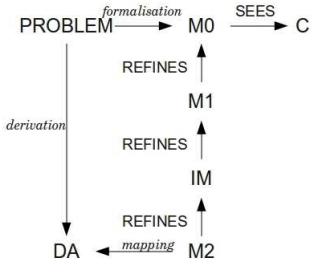
disjunction of the guards of the events of the concrete model. Relative deadlock freeness is now easily formalized as the following proof obligation (4):

$$I(x) \ \wedge \ J(x,y) \ \wedge \ \mathsf{grds}(AM) \ \Rightarrow \ \mathsf{grds}(CM)$$

When one refines a model, one can either refine an existing event by strengthening the guard and/or the before-after predicate (effectively reducing the degree of non-determinism), or add a new event in order to refine the $skip$ event. The feasibility condition is crucial for avoiding possible states which have no successor; for instance, the division by zero. Furthermore, such refinement guarantees that a set of traces of the refined model contains (up to stuttering) traces of the resulting model. The basic foundations of the Event B modeling language along with several case studies are availbale in [1, 6]. The language of *generalized substitutions* is very rich and allows us to express any relation between states in a set-theoretical context. The expressive power of the language leads to require helps for writing relational specifications and this is why we should provide proof-based patterns for assisting the development of Event B models.

B. General Schema for Refinement

The correct-by-construction approach is based on the use of refinement and to introduce new features in the formal models. The methodology is simply described by the following diagram, which advocates different steps for producing a distributed algorithm using the correct-by-construction approach.



- The context *C* states properties of graphs.
- The machine *M0* expresses the problem to solve by a set of events stating a relation between initial and final states, for instance, the computation of a correct snapshot.
- The refinement of *M0* into *M1* expresses that the machine *M1* expresses the inductive property allowing to express the computation of the snapshot by each node.
- The refinement of *M1* by *IM* prepares the localisation phase and may require more than one refinement step.
- The next refinement of *IM* is a refinement for producing a set of events corresponding to the localisation of information.
- *DA* is derived from the *M2*; *mapping* checks that *M2* can be translated into a distributed programming language.

However, we consider a more general schema for developing the snapshot problem, since the snapshot problem is solved by an algorithm which is able to compute the current distributed state. Next subsection starts the refinement process by introducing the first refinement related to the observation of the snapshot.

C. Introducing the OBSERVATION model

The OBSERVATION model *refines* the SYSTEM model and introduces the functionality, which is required by the snapshot problem: *to compute a snapshot*. It does not explain how to compute it but what it should compute.

MACHINE OBSERVATION
...
EVENT Snapshot
ANY
$acut$
WHERE
$grd1 : acut \in P \to \mathbb{N}$
$grd2 : \forall p, q, i, j, m \cdot$
$\left(\begin{array}{l} p \in P \wedge q \in P \wedge m \in M \wedge p \neq q \\ \wedge i \in dom(h(p)) \wedge j \in dom(h(q)) \wedge j \leq acut(q) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in s \wedge (p \mapsto q \mapsto j) \mapsto m \in r \end{array} \right)$
$\Rightarrow i \leq acut(p)$
$grd3 : \forall p \cdot p \in P \Rightarrow acut(p) \in dom(h(p))$
THEN
$act1 : cut := acut$
END
...

The event snapshot states that a consistent cut (obtained in *one-shot*), namely $acut$, is assigned to cut: a moving message is not allowed to be part of the snapshot, if origin of the message is outside of the cut and its destination is inside of the cut. The event expresses the intention to specify the required solution. Further refinements are necessary for introducing the inductive process leading to a consistent cut. Others events are related to the previous models, which are indicated by dots. Due to space limitations, we have given sketch of the modeling. A detailed formal development is available[1].

V. ARCHITECTURE OF THE DESIGN

Figure 2 presents the complete formal development, which starts from SYSTEM and NETWORK and progressively leads to the OBSERVATION and PROCESS. We describe the model PROCESS which provides the underlying computing process to produce a consistent snapshot.

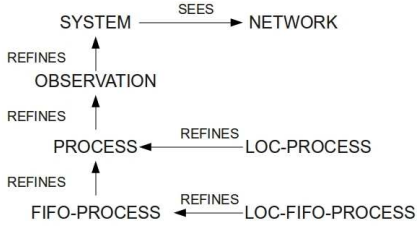


Figure 2. General Architecture of the Design

[1]http://www.loria.fr/~andriami/snapshot-pdf/project.html

A. Computing a snapshot

The PROCESS model (see Fig.3) refines the OBSERVATION model, and presents the construction of a correct snapshot ($pcut$) *step-by-step*. A control message ($marker$) is introduced along with events to separate pre and post-snapshot messages for describing the development steps of the snapshot algorithm :

MACHINE PROCESS
...
INVARIANTS
...
$A : \forall p, q, i, j, m \cdot \left(\begin{array}{l} p \in P \wedge q \in P \wedge m \in M \wedge p \neq q \\ \wedge p \in dom(pcut) \wedge q \in dom(pcut) \wedge j \leq pcut(q) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in s \wedge (p \mapsto q \mapsto j) \mapsto m \in r \\ \Rightarrow i \leq pcut(p) \end{array} \right)$
$B : \forall p, q, i, j, m \cdot \left(\begin{array}{l} p \in P \wedge q \in P \wedge m \in M \wedge p \neq q \\ \wedge p \in dom(pcut) \wedge q \notin dom(pcut) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in s \wedge (p \mapsto q \mapsto j) \mapsto m \in r \\ \Rightarrow i \leq pcut(p) \end{array} \right)$
$C : \forall p, q, i, m \cdot \left(\begin{array}{l} p \in P \wedge q \in P \wedge m \in M \wedge p \neq q \\ \wedge p \in dom(pcut) \wedge q \notin dom(pcut) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in s \wedge m \notin mark(q \mapsto p) \\ \Rightarrow i \leq pcut(p) \end{array} \right)$
...
EVENTS
...
EVENT StartingSnapshot ...
EVENT ProgressingSnapshot ...
EVENT Snapshot ...
EVENT SendingPreSnapshot REFINES Sending
ANY
$achan, m, p, q, i$
WHERE
$grd1 : achan \in C \wedge m \in M \wedge i \in \mathbb{N} \wedge p \in P \wedge q \in P$
$grd2 : \forall chan \cdot chan \in C \Rightarrow m \notin c(chan)$
$grd3 : p \mapsto q = achan$
$grd4 : o(p) + 1 \notin dom(h(p))$
$grd5 : m \notin ran(s) \wedge i = o(p) + 1$
$grd6 : p \mapsto q \mapsto i \notin dom(s)$
$grd7 : p \notin dom(pcut)$
THEN
$act1 : c(achan) := c(achan) \cup \{m\}$
$act2 : o(p) := o(p) + 1$
$act3 : h(p) := h(p) \cup \{o(p) + 1 \mapsto (p \mapsto q)\}$
$act4 : s := s \cup \{(p \mapsto q \mapsto i) \mapsto m\}$
END
EVENT SendingPostSnapshot ...
EVENT ReceivingPreSnapshotMessages ...
EVENT ReceivingPostSnapshotMessages ...
EVENT ReceivingBeforeTheSnapshot ...
...
END

Figure 3. The PROCESS Machine

- StartingSnapshot: A node ($special$) starts to build of the snapshot. The node ($special$) saves its local state. It begins to record the incoming messages ($marker$) and finally, this node ($special$) sends a message ($marker$) to all of its neighbouring nodes.
- ProgressingSnapshot: A node (i) receives a message ($marker$) from the neighbouring node and it begins to record all the incoming messages. If the node (i) receives all the messages before sending the message ($marker$), it records the local state and transmits the message ($marker$) to its neighbours.
- Snapshot: All the nodes have received a message ($marker$). For all the nodes, the messages sent to them before a message ($marker$), have been received. Finally, the global state of the distributed system is saved.

The model also introduces a set of properties for describing the consistency of the cut:

(A) If a message m is sent by a process (p) at a time (i), and received by a process (q) at a time (j) before the snapshot, then the time (i) belongs to the past of the cut.
(B) If a message m is sent by a process (p) (which has already performed a local cut) at the time (i), received by a process (q) (which has not yet performed a local cut) at a time (j), then the time (i) belongs to the past of the cut.
(C) If a message m has been sent by a process (p) to process (q) at a time (i) (before the receiving of a message ($marker$) by the process (p)), then the time (i) belongs to the past of the cut.

The events Sending and Receiving are refined to distinguish pre-snapshot messages and/or activities from their post-snapshot counterparts:

- SendingPreSnapshot: This event describes the sending of a message (m) by a process (p), before the local cut of the process (p).
- SendingPostSnapshot: This event presents the sending of the message (m) by the process (p), which follows the local cut of the process (p). The message (m) is marked as being sent after the local cut of the process(p).
- ReceivingPreSnapshotMessages: This event demonstrates the receiving of the message (m) (sent by the process (p) before the local cut of the process (p)) by a process (q), after receiving a message ($marker$) by the process (q). The incoming message (m) is recorded by the process (q).
- ReceivingPostSnapshotMessages: This event shows the receiving of a message (m) (sent by a process (p) after the local cut of the process (p)) by a process (q), after the process (q) has performed a local cut.
- ReceivingBeforeTheSnapshot: This event describes the receiving of a message (m) (sent by a process (p) before the local cut of the process (p)) by a process (q), before the process (q) performs a local cut.

B. Deriving Snapshot Algorithms

1) The Lai and Yang Algorithm: The Lai and Yang algorithm [13] is a two-phases protocol: either (A) one special process (called $initiator$) initiates the snapshot, or (B) another process among non-initiator processes extends the snapshot. Due to their similarities, we will focus on phase (A), depicted by the following steps:

```
process (initiator) :
  step 1: record local state;
  step 2: snapshot := 1;
  step 3: begin to record incoming pre-snapshot messages;
  step 4: to send a message : <message, snapshot>;
```

Details of the two possible phases are described by the model PROCESS, an abstract model of the Lai and Yang algorithm [13]: channels between processes are represented by sets of messages; however a message (m) is extended by a bit, which determines either if the message is pre or post-snapshot. The bit is 1, when the predicate $m \in mark(c)$ holds. The model LOC-PROCESS, refining PROCESS, localizes informations and describes a model for the Lai and Yang algorithm. We can identify, in the LOC-PROCESS model, events representing the phases (A) and (B) of the Lai and Yang algorithm:

EVENT StartingSnapshot REFINES StartingSnapshot
ANY
nsm
$ncstate$
nm
WHERE
$grd1 : initiator \notin dom(pcut)$
$grd2 : ncstate \in C \nrightarrow \mathbb{P}(M)$
$grd3 : ncstate = cstate \cup \{d \mapsto \varnothing | d \in C \wedge prj2(d) = initiator\}$
$grd4 : nsm \subseteq C$
$grd5 : nm \in C \nrightarrow \mathbb{P}(M)$
$grd6 : nsm = send\_mark \cup \{d | d \in C \wedge prj1(d) = initiator\}$
$grd7 : nm = \{d \mapsto \varnothing | d \in nsm\}$
THEN
$act1 : pstate(initiator) := l(initiator)$
$act2 : pcut(initiator) := o(initiator)$
$act3 : cstate := ncstate$
$act4 : send\_mark := nsm$
$act5 : mark := nm$

The actions of this event can be associated with the steps of phase (A) :

- $act1$ models **step 1**: the process ($initiator$) records its local state.
- $act2$ represents **step 2**: the process ($initiator$) takes a local snapshot.
- $act3$ indicates that the process ($initiator$) will record all pre-snapshot incoming messages (**step 3**).
- Finally, $act4$ and $act5$ match **step 4**: the process ($initiator$) indicates that all outgoing messages will be labelled with the bit 1.

We can see that the two phases (A) and (B) are modelled, respectively, by the events StartingSnapshot and ProgressingSnapshot. The other events do not describe parts of the Lai and Yang algorithm; they depict activities of the processes and the network (communications, computations, etc.).

2) The Chandy and Lamport Algorithm: The Chandy and Lamport algorithm [9] uses a mechanism of coloring and propagation of a red color from a white one. A white message occurs before a snapshot and a red message occurs after the snapshot. We split the two kinds of messages using a variable *mark*, indicating, whether or not messages ($marker$) have been sent by processes. The abstract model FIFO-PROCESS of this algorithm refines the model PROCESS: it is an abstract model of the Lai and Yang algorithm. Behaviours of the model FIFO-PROCESS correspond to behaviours of the model PROCESS, thanks to refinement. However, the FIFO-PROCESS introduces new features: the separation between the pre and post-snapshot messages is implemented by a FIFO communication mechanism. Channels between nodes are transformed from sets of messages to FIFO queues. Because of the clear distinction between the pre and post-snapshot phases, the bit of membership defined in the Lai and Yang algorithm can be removed; which means that the messages are less complex. However, we can observe that a strong constraint is added: in the Chandy and Lamport algorithm, FIFO communication channels are mandatory. The LOC-FIFO-PROCESS model refines the FIFO-PROCESS model: the LOC-FIFO-PROCESS model localizes events and is producing the algorithmic form of the Chandy and Lamport algorithm.

VI. Discussion, Conclusion and Future Work

The snapshot algorithm identifies global states in a distributed system. The result of our works on the snapshot problem is the discovery of a generic architecture which allows the derivation of various algorithms. The model SYSTEM describes a distributed system and the activities of its processes (computations, communications, etc.). This model is generic: computations, activities, etc. can be made more specific, according to the peculiarities of studied systems. The model SYSTEM is refined by a model OBSERVATION, which introduces the notion of *snapshot*: an event models the *global snapshot* of the distributed system. The development of the snapshot is organised from the model called PROCESS, which expresses the underlying computation process and can be refined into several other algorithms. The key idea is to separate the pre-shots and the post-shots and the solution depends on assumptions on channels and messages: the $mark$ variable is either a marker for a bit or a marker for *fifo* channels. The complexity of the development is measured by the number of proof obligations which are automatically/manually discharged (see table VI). The main difficulty of the development was the expression of a consistent snapshot in the machine PROCESS, therefore the establishment of the refinement relation between PROCESS and the machine OBSERVATION. A set of invariants (A,B,C) of the machine PROCESS (Fig.3) were the keys of the development, where the generated proof obligations were quite difficult to discharge. Moreover, the snapshot algorithm is supposed to work while another process SYSTEM is working; SYSTEM is a model for another distributed system and the snapshot algorithm is an implementation of the observation of the current system. Contrary to the verification by theorem provers [18], our work provides an architecture for developing the snapshot algorithm using essential safety properties together with a formal proof that asserts its correctness.

Model	Total	Auto		Interactive	
NETWORK	10	10	100%	0	0%
SYSTEM	42	36	85.71%	6	14.29%
OBSERVATION	35	18	51.43%	17	48.57%
PROCESS	95	44	46.32%	51	53.68%
Total	182	108	59.34%	74	40.66%

Table I
SUMMARY OF PROOF OBLIGATIONS

In this paper, we have experimented on fixed networks. As a part of our future efforts we consider the global family of snapshot algorithms to give a very precise description of different solutions and to link between these algorithms, as we notice that the algorithm of Chandy and Lamport is obtained from the algorithm of Lai and Yang by adding a FIFO communication. Moreover, we plan to integrate the snapshot algorithm with complex distributed systems like mobile networks.

REFERENCES

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

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

[3] R.-J. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, 1988.

[4] R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.

[5] R.-J. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.

[6] D. Bjorner and M. C. Henson, editors. *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, 2007.

[7] D. Cansell and D. Méry. Designing old and new distributed algorithms by replaying an incremental proof-based development. In J.-R. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2009.

[8] K. M. Chandy, B. Go, S. Mitra, and J. White. Towards verified distributed software through refinement of formal archetypes. In *IFIP Working Conference on Verified Software: Workshop on Experiments*, October 2008.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[10] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[11] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. The TLA$^+$ proof system: Building a heterogeneous verification platform. In A. Cavalcanti, D. Déharbe, M.-C. Gaudel, and J. Woodcock, editors, *International Conference on Theoretical Aspects of Computing - ICTAC 2010*, volume 6255 of *Lecture Notes in Computer Science*, page 44, Brazil Natal, 2010. Springer. The original publication is available at www.springerlink.com.

[12] E.M.Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, december 1999. ISBN 0-262-03270-8.

[13] T.-H. Lai and T. H. Yang. On distributed snapshots. *Inf. Process. Lett.*, 25(3):153–158, 1987.

[14] L. Lamport. *Specifying Systems: The TLA$^+$+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[15] G. T. Leavens, J.-R. Abrial, D. S. Batory, M. J. Butler, A. Coglio, K. Fisler, E. C. R. Hehner, C. B. Jones, D. Miller, S. L. P. Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 221–236. ACM, 2006.

[16] B. L. Massingill and K. M. Chandy. Parallel program archetypes. In *In Proceedings of the Scalable Parallel Library Conference*, pages 1–9, 1997.

[17] O. A. Mohamed, C. Muñoz, and S. Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.

[18] K. Ogata and P. T. T. Huyen. Specification and model checking of the chandy & lamport distributed snapshot algorithm in rewriting logic. In *ICFEM 2012*, 2012.

[19] S. Owre and N. Shankar. A brief overview of pvs. In Mohamed et al. [17], pages 22–27.

[20] Project RODIN. Rigorous open development environment for complex systems. http://www.eventb.org/, 2004-2010.

[21] G. Tel. *Introduction to Distributed Algorithms*. Cambridge Unversity Press, 1994.

[22] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In Mohamed et al. [17], pages 33–38.

[23] Z. Yang and T. A. Marsland. Global snapshots for distributed debugging: An overview. Technical report, Computing Science Department, University of Alberta, 1992.