Computer Science and Information Systems 11(1):251–270 DOI: 10.2298/CSIS130122007A

Revisiting Snapshot Algorithms by Refinement-based Techniques

Manamiary Bruno Andriamiarina[1], Dominique Méry[1], and Neeraj Kumar Singh[2]

[1] Université de Lorraine, LORIA
Vandœuvre-les-Nancy, France
{Manamiary.Andriamiarina, Dominique.Mery}@loria.fr
[2] McMaster Centre for Software Certification, McMaster University
Hamilton, Ontario, Canada
singhn10@mcmaster.ca

Abstract. The snapshot problem addresses a collection of important algorithmic issues related to distributed computations, which are used for debugging or recovering distributed programs. Among existing solutions, Chandy and Lamport have proposed 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 correct distributed algorithms. Moreover, we demonstrate how other distributed algorithms can be revisited. A consequence is to provide a fully mechanized proof of the resulting distributed algorithms.

Keywords: Distributed algorithms, correctness-by-construction, refinement, snapshot, verification.

1. 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 [13, 28, 30]. 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 [14, 15] and theorem prover [14, 25, 29]. The correct-by-construction paradigm [18] 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, 6, 20, 21]. 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 [9] 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[19] introduce *archetypes* for facilitating parallel program design; more recently, Chandy et al [12] 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 [13], the algorithm of Lai and Yang [16] or the algorithm of Morgan [23]. Finally, the snapshot problem is already considered as a case study for illustrating the strength of rewriting logic [24] and we think that our development may help a reader to understand the behavioral theory of snapshot algorithms.

Our Contribution. This paper contributes to demonstrate semantical relationships existing between various snapshot algorithms (algorithms of Chandy and Lamport [13], Lai and Yang [16], Morgan [23]) with the refinement of models. We start with an abstract initial specification of the snapshot problem and we enrich this specification gradually by a progressive and incremental refinement. Several refinement steps allow to capture the complete and desired behaviour of snapshot algorithms. The refinement of models is the key element allowing preservation of properties between the levels of abstraction.

Moreover, we propose an architecture based on the *correction-by-construction* paradigm, for conceiving algorithms dedicated to observation of global states of distributed systems. This capture of a global state of a distributed system is introduced in the architecture by a model OBSERVATION. Our architecture is reusable and extendable since algorithms can be conceived or studied by refining either the OBSERVATION model or more concrete models (algorithms) provided in our architecture.

Organization of the Paper. The paper is organized as follows. Section 2 presents related works on the design of distributed snapshots. Section 3 defines the snapshot problem in distributed systems. Section 4 introduces notations of Event B and the formal activities of a global system. Section 5 presents refinement-based development of the snapshot algorithm, where we describe the OBSERVATION model for stating *what* we have to compute. Section 6 introduces the computation of a snapshot in the ASYNC-PROCESS and SYNC-PROCESS models, which simulate the OBSERVATION model. The global architecture of the refinement-based design is similar to the classical distributed algorithms [13, 16]. Section 6 also compares the formal modelling of the snapshot algorithms [13, 16, 23]. Section 7 concludes this paper along with the future work.

2. Related Works

Several literatures [3, 10, 28, 30] report works on the design of algorithms for the observation of distributed systems. Jaggi et al. [3] have proposed a snapshot algorithm (Distributed Snapshot Algorithm for MANETs) DSAM, which is derived from the algorithm of Chandy and Lamport, for dealing with snapshots in a mobile ad-hoc network. Chalopin et al. [10] have produced an algorithm combining the Chandy and Lamport and the SSP [27] algorithms for detecting the termination of the snapshot. Yang and Marsland [30] have elaborated debugging frameworks for distributed systems using various existing snapshot algorithms (Venkatesan, Lai and Yang, Li, Radhakrishnan and Venkatesh, Spezialetti and Kearns Algorithm, Morgan). The main motivation of these works is the need of *observing* global states of distributed systems, for identifying stable properties such as termination and deadlock [10], creating breakpoints for recovery, debugging systems [30], or taking into account non-fixed/mobile networks [3]. These works also present elements for proving the correctness of the designed algorithms. However, these elements are only partial proofs and do not assert completely the correctness and safe behaviour of the algorithms.

All these works present new algorithms, based on well-known classical ones (Chandy and Lamport, Lai and Yang, Morgan, etc), for the observation of distributed systems. However, the semantical link between the new and classical algorithms is not formally established. These works also focus on conceiving correct algorithms able to produce correct/consistent snapshots. Yet, as far as we know, the existing works are mainly based on simulation, which cannot guarantee the safe behavior. In this study, we use the formal techniques to specify the snapshot algorithms to make sure of the safe behavior, correctness and consistency using safety properties.

3. 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 [28] 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 steps for deriving three well-known snapshot algorithms [13, 16, 23] using proof-assisted stepwise development.

4. 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, 8] borrows features from formal modeling languages like UNITY [11], TLA$^+$ [17], action systems [4, 5]; 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 [26].

4.1. Modelling Actions Over States

The event-driven approach [1, 8] 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 [26].

Modifications over state variables are stated by events. An event e has two main parts: a *guard* grd(e), which is a predicate built on the state variables, and an *action*, which is a generalized substitution. An event e can take one of the three normal forms described in the table 1 and is associated with a before-after predicate $BA(e)(x, x')$, 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 Table 1).

Event e	Before-after Predicate BA(e)(x,x')	Guard grd(e)(x)
BEGIN $x :\|(P(x,x'))$ END	$P(x,x')$	$TRUE$
WHEN $G(x)$ THEN $x :\|(Q(x,x'))$ END	$G(x) \wedge Q(x,x')$	$G(x)$
ANY t WHERE $G(t,x)$ THEN $x :\|(R(x,x',t))$ END	$\exists t \cdot (\, G(t,x) \wedge R(x,x',t)\,)$	$\exists t.G(t,x)$

	Proof obligation
(INV1)	$Init(x) \Rightarrow I(x)$
(INV2)	$I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$
(FIS)	$I(x) \wedge \mathsf{grd}(e)(x) \Rightarrow \exists y.BA(e)(x,y)$

Table 1. Event B events and proof obligations

Proof obligations (INV 1 and INV 2) are produced by the tool RODIN [26] 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.

4.2. Describing the Network and its Activities

```
CONTEXT NETWORK
SETS
  P, M, PStates
CONSTANTS
  C  ...
AXIOMS
  axm1 : P ≠ ∅
  axm2 : M ≠ ∅
  axm3 : Pstates ≠ ∅
  axm4 : C ⊆ (P × P) \ id
  ...
END
```

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.

4.3. 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.

The defined variables for this model SYSTEM are as follows:

```
MACHINE SYSTEM
  ...
  EVENT InternalLocal
  ...
  EVENT InternalMessage
  ...
  EVENT Sending
  ...
  EVENT Receiving
  ...
  ...
END
```

- o associates each process to the timestamp of its last operation.
- l describes the local state of each process.
- h contains the traces of the activities (*history*) for each process.
- $chan$ presents a set of messages that circulates inside channels.
- $store$ depicts a set of messages that is stored by each process.
- $send$ models the sending messages that are sent by the processes.

The activities of the distributed system depicted in the model SYSTEM are as follows:

a) *Internal* operations modify states and variables local to nodes. These activities are modelled by the following events:

```
EVENT InternalLocal
  ANY
    p, ns
  WHERE
    grd1 : p ∈ P ∧ ns ∈ PStates
  THEN
    act1 : l(p) := ns
    act2 : o(p) := o(p) + 1
    act3 : h(p) := h(p) ∪ {o(p) + 1 ↦ (p ↦ p)}
```

```
EVENT InternalMessage
  ANY
    p, m
  WHERE
    grd1 : p ∈ P ∧ m ∈ M ∧ m ∈ store(p)
  THEN
    act1 : o(p) := o(p) + 1
    act2 : h(p) := h(p) ∪ {o(p) + 1 ↦ (p ↦ p)}
    act3 : store(p) := store(p) \ {m}
```

- InternalLocal demonstrates the modification of a local state of a process (p). A new state (ns) is chosen non-deterministically for the process (p).
- InternalMessage models the modification of the local set of messages of a process (p).
  The process (p) deletes a message (m) from its local set of messages ($store(p)$).

b) *External* operations involve different nodes. These operations are described by the following events:

```
EVENT Sending
  ANY
    p, m, c
  WHERE
    grd1 : p ∈ P ∧ m ∈ M ∧ c ∈ C
    grd2 : prj1(c) = p ∧ m ∉ send ∧ m ∉ chan(c)
  THEN
    act1 : send := send ∪ {m}
    act2 : chan(c) := chan(c) ∪ {m}
    act3 : o(p) := o(p) + 1
    act4 : h(p) := h(p) ∪ {o(p) + 1 ↦ c}
```

```
EVENT Receiving
  ANY
    q, m, c
  WHERE
    grd1 : q ∈ P ∧ m ∈ M ∧ c ∈ C
    grd2 : prj2(c) = q ∧ m ∈ chan(c)
  THEN
    act1 : chan(c) := chan(c) \ {m}
    act2 : o(q) := o(q) + 1
    act3 : h(q) := h(q) ∪ {o(q) + 1 ↦ c}
    act4 : store(q) := store(q) ∪ {m}
```

- Sending defines the sending of a message (m) by a process (p), through a channel (c). The message (m) has not yet been sent by the process (p) and is not circulating inside the channel (c). Therefore, the process (p) is able to send (m) through the channel (c) connecting the process (p) to another process.

- Receiving presents the receiving of a message (m) by a process (q), via a channel (c). The message (m) is inside the channel (c) leading to the process (q): The message (m) is removed from the channel (c) and is stored by the process (q).

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). A new step expresses the observation of the current system by another process which is defined by a refinement[3] of the current model. In the next section, we define the refinement and apply it for the observation.

5. Incremental Proof-Based Development

5.1. 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* [18] 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):

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

[3] $\oplus$: add elements to a model; $\ominus$: remove elements from a model

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 available in [1, 7]. 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.

5.2. General Schema for Refinement

The correct-by-construction approach is based on the use of refinement and on introducing 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.

$$\begin{tikzcd}
\text{PROBLEM} \arrow[r, "formalization"] \arrow[ddd] & \text{M0} \arrow[r, "SEES"] & \text{C} \\
& \text{M1} \arrow[u, "REFINES"'] & \\
& \text{IM} \arrow[u, "REFINES"'] & \\
\text{DA} & \text{M2} \arrow[l, "mapping"] \arrow[u, "REFINES"'] &
\end{tikzcd}$$

- 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* presents 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.

5.3. 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.

A set of new variables is introduced to model the required behaviour as follows:

- Two variables s and r are defined for ordering the sending and receiving of messages:
  - s associates sent messages with channels and timestamps. The variable helps the users to determine the channel in which a message is sent and the time of the sending operation.
  - r associates received messages with channels and timestamps. This variable indicates the channel from which a message is received by a process and the time of the receiving operation.
- cut contains the result of the snapshot.

The refined versions of the events Sending and Receiving are modified to take the variables r and s into account:

- Sending uses the variable s to indicate the channel (c) in which a message (m) is sent and the sending time.
- Receiving records a message (m) and receiving time from the channel (c) into the variable r.

MACHINE OBSERVATION
. . .
EVENT Sending REFINES Sending
  . . .
  THEN
    . . .
    $\oplus\ :\ s := s \cup \{c \mapsto o(p) + 1 \mapsto m\}$
  END

EVENT Receiving REFINES Receiving
  . . .
  THEN
    . . .
    $\oplus\ :\ r := r \cup \{c \mapsto o(q) + 1 \mapsto m\}$
  END

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

A new event Snapshot models an abstraction of the snapshot procedure and 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 a sketch of the modeling. A detailed formal development is available[4].

6. Architecture of the Design

Figure 1 presents the complete formal development, which starts from SYSTEM and NETWORK and progressively leads to the OBSERVATION. The models ASYNC-PROCESS and SYNC-PROCESS are derived from the model OBSERVATION and present two different ways of computing a consistent snapshot: ASYNC-PROCESS describes an asynchronous computation of a snapshot [13, 16], whereas SYNC-PROCESS depicts a synchronous way of constructing a snapshot [23].

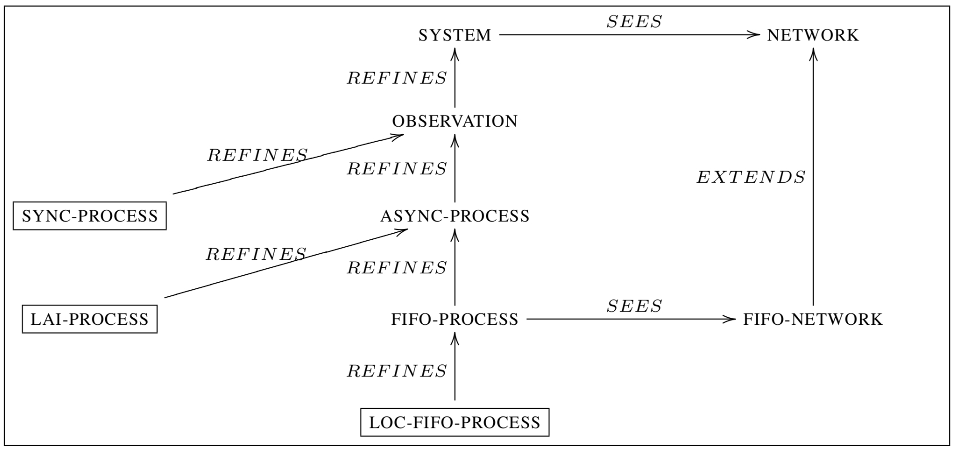

Fig. 1. General Architecture of the Design

It should be noted that SYNC-PROCESS and ASYNC-PROCESS model respectively the algorithms of Morgan [23] and Lai and Yang [16]. The model LAI-PROCESS presents a more concrete version of the snapshot algorithm of Lai and Yang [16]. A context FIFO-NETWORK extends NETWORK and adds elements of FIFO queues and communications ordering to the network. The model of the algorithm of Chandy and Lamport [13] (FIFO-PROCESS) is refined from the model ASYNC-PROCESS and uses the context FIFO-NETWORK. The model LOC-FIFO-PROCESS is a refinement of FIFO-PROCESS and defines a more concrete and local version of the snapshot algorithm of Chandy and Lamport [13].

6.1. Computing a Synchronous Snapshot

The SYNC-PROCESS model (see Fig.2) is a refinement of OBSERVATION, and defines the synchronous construction of a correct snapshot ($pcut$) *gradually*: an external parameter (e.g. a global clock) triggers the snapshot procedure for all the processes, at the same time. More precisely, the SYNC-PROCESS model describes the steps of the snapshot algorithm of Morgan [23], which is based on the availability of a global time for all the processes.

New variables are introduced:

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

MACHINE SYNC-PROCESS
...
INVARIANTS
...

$$A : \forall p, q, i, j, m\cdot \begin{pmatrix} p \in P \wedge q \in P \wedge m \in M \wedge p \neq q \\ \wedge p \in dom(pcut) \wedge q \in dom(pcut) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in s \wedge (p \mapsto q \mapsto j) \mapsto m \in r \\ \wedge j \leq pcut(q) \end{pmatrix} \Rightarrow i \leq pcut(p)$$

...
EVENTS
...
EVENT Tick ...
EVENT ProcessingSnapshot ...
EVENT Snapshot ...
EVENT SendingBeforeCut ...
EVENT SendingAfterCut ...
EVENT ReceivingPreCutMessages ...
EVENT ReceivingPostCutMessages ...
EVENT ReceivingBeforeCut ...
...
END

Fig. 2. The SYNC-PROCESS Machine

- $mark\_m$ contains a set of messages that is sent after the snapshot.
- tm represents the global time.
- $cstate$ records, for each process, the incoming *pre-snapshot* messages.
- $pstate$ contains, for each process, the state of the process during the snapshot.
- $pcut$ is an intermediate variable presenting the construction of the snapshot.

The invariant (A) (see Fig.2) defines constraints on these variables and states the consistency of the snapshot: 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.

Events describe the computation of the snapshot :

- Tick models the flow of time.

EVENT Tick
WHEN
$grd1 \;:\; tm = FALSE$
THEN
$act1 \;:\; tm := TRUE$

We model the fact that a predefined global time (t) (for triggering the snapshot) has been reached, by setting the value of tm to $TRUE$.

EVENT ProcessingSnapshot
ANY $ncstate$
WHERE
$grd1 \;:\; ncstate \in C \rightarrow \mathbb{P}(M) \wedge tm = TRUE$
$grd2 \;:\; ncstate = \{k \mapsto \varnothing | k \in C\}$
THEN
$act1 \;:\; cstate := ncstate$
$act2 \;:\; pcut := o$
$act3 \;:\; pstate := l$

EVENT Snapshot REFINES *Snapshot*
WHEN
$grd1 \;:\; dom(pcut) = P$
$grd2 \;:\; tm = TRUE$
$grd3 \;:\; \forall p\cdot p \in C \Rightarrow chan(p) \subseteq mark\_m$
WITH
$acut \;:\; acut = pcut$
THEN
$act1 \;:\; cut := pcut$

- ProcessingSnapshot demonstrates the simultaneous local snapshots of all the processes. When the global time (t) (for triggering the snapshot) has been reached, all the processes records their local states and begin to save incoming *pre-snapshot* messages.
- Snapshot describes the global snapshot. An abstract parameter $acut$ (modelling the global state of the system) of the event is removed and replaced by concrete variable $pcut$. This new variable $pcut$ represents the global state of the system and its value is computed in previous event ProcessingSnapshot. When all the processes have recorded their local states, as well as all the incoming *pre-snapshot* messages, the global state ($pcut$) of the system is saved into the variable cut.

The events Sending and Receiving are refined and their refinements describe the *pre-snapshot* and *post-snapshot* activities of the system:

```
EVENT SendingBeforeCut REFINES Sending
  ...
  WHERE
    ...
    ⊕ : tm = FALSE
    ...
```

```
EVENT SendingAfterCut REFINES Sending
  ...
  WHERE
    ⊕ : p ∈ dom(pcut)
  THEN
    ⊕ : mark_m := mark_m ∪ {m}
```

- SendingBeforeCut: This event models the sending of messages before the global time (t) for processing the snapshot is reached. The actions of this event are similar to the actions of the abstract event Sending.
- SendingAfterCut: This event demonstrates the sending of messages after the local snapshot of a process (p). The message (m) is *marked* as being sent by the process (p) after the local cut.
- ReceivingPreCutMessages: This event presents the receiving of *pre-snapshot* messages by a process (q), after a local cut.

```
EVENT ReceivingPreCutMessages REFINES Receiving
  ...
  WHERE
    ...
    ⊕ : q ∈ dom(pcut)
    ⊕ : m ∉ mark_m
  THEN
    ...
    ⊕ : cstate(c) := cstate(c) ∪ {m}
```

The message (m) received by the process (q) is recorded as a *pre-snapshot* message.

```
EVENT ReceivingPostCutMessages REFINES Receiving
  ...
  WHERE
    ⊕ : q ∈ dom(pcut)
    ⊕ : m ∈ mark_m
  THEN
    ⊕ : mark_m := mark_m \ {m}
```

```
EVENT ReceivingBeforeCut REFINES Receiving
  ...
  WHERE
    ...
    ⊕ : tm = FALSE
    ⊕ : m ∉ mark_m
    ...
```

- ReceivingPostCutMessages: This event expresses the receiving of marked *post-snapshot* messages by a process (q), after a local cut. The message (m) received by the process (q) is removed from the set ($mark\_m$) of marked *post-snapshot* messages.
- ReceivingBeforeCut: This event models the receiving of messages before the global time (t) for processing the snapshot is reached. The actions of this event are the same as the actions of the abstract event Receiving.

The following sections describe the derivation of other algorithms for computing snapshots, in a complete local manner, without any global mean [13, 16].

6.2. Computing an Asynchronous Snapshot

The ASYNC-PROCESS model (see Fig.3) refines the OBSERVATION model, and presents the asynchronous 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 ASYNC-PROCESS
  ...
INVARIANTS
  ...

$$A : \forall p, q, i, j, m \cdot \begin{pmatrix} p \in P \land q \in P \land m \in M \land p \neq q \\ \land p \in dom(pcut) \land q \in dom(pcut) \land j \leq pcut(q) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \land (p \mapsto q \mapsto j) \mapsto m \in r \end{pmatrix} \Rightarrow i \leq pcut(p)$$

$$B : \forall p, q, i, j, m \cdot \begin{pmatrix} p \in P \land q \in P \land m \in M \land p \neq q \\ \land p \in dom(pcut) \land q \notin dom(pcut) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \land (p \mapsto q \mapsto j) \mapsto m \in r \end{pmatrix} \Rightarrow i \leq pcut(p)$$

$$C : \forall p, q, i, m \cdot \begin{pmatrix} p \in P \land q \in P \land m \in M \land p \neq q \\ \land p \in dom(pcut) \land q \notin dom(pcut) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \land m \notin mark\_m \end{pmatrix} \Rightarrow i \leq pcut(p)$$

  ...
EVENTS
  ...
  EVENT StartingSnapshot ...
  EVENT ProgressingSnapshot ...
  EVENT Snapshot ...
  EVENT SendingBeforeCut ...
  EVENT SendingAfterCut ...
  EVENT ReceivingPreCutMessages ...
  EVENT ReceivingPostCutMessages ...
  EVENT ReceivingBeforeCut ...
  ...
END

Fig. 3. The ASYNC-PROCESS Machine

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

EVENT ProgressingSnapshot
  ANY
    $i, c, nsm, ncstate$
  WHERE
    $grd1 : i \in P \land c \in C$
    $grd2 : nsm \subseteq C$
    $grd3 : ncstate \in C \nrightarrow \mathbb{P}(M)$
    $grd4 : i = prj2(c) \land i \notin dom(pcut)$
    $grd5 : c \in send\_mark \land chan(c) \subseteq mark\_m$
    $grd6 : nsm = send\_mark \cup \{d | d \in C \land prj1(d) = i\}$
    $grd7 : ncstate = cstate \cup \{d \mapsto \varnothing | d \in C \land prj2(d) = i\}$
  THEN
    $act1 : pcut(i) := o(i)$
    $act2 : pstate(i) := l(i)$
    $act3 : cstate := ncstate$
    $act4 : send\_mark := nsm$

- StartingSnapshot: A node ($initiator$) starts to build of the snapshot. The node ($initiator$) 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.

```
EVENT Snapshot REFINES Snapshot
  WHEN
    grd1 : dom(pcut) = P
    grd2 : send_mark = C
    grd3 : ∀p·p ∈ C ∧ p ∈ send_mark ⇒ chan(p) ⊆ mark_m
  WITH
    acut : acut = pcut
  THEN
    act1 : cut := pcut
```

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:

```
EVENT SendingBeforeCut REFINES Sending
  ...
  WHERE
    ...
    ⊕ : p ∉ dom(pcut)
    ...
```

```
EVENT SendingAfterCut REFINES Sending
  ...
  WHERE
    ⊕ : p ∈ dom(pcut)
  THEN
    ⊕ : mark_m := mark_m ∪ {m}
```

– SendingBeforeCut: This event describes the sending of a message (m) by a process (p), before the local cut of the process (p). The actions of this event are similar to the actions of the *"normal"* event Sending.
– SendingAfterCut: 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).
– ReceivingPreCutMessages: 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).

```
EVENT ReceivingPreCutMessages REFINES Receiving
  ...
  WHERE
    ...
    ⊕ : q ∈ dom(pcut)
    ⊕ : m ∉ mark_m
  THEN
    ...
    ⊕ : cstate(c) := cstate(c) ∪ {m}
```

The incoming message (m) is recorded by the process (q).

```
EVENT ReceivingPostCutMessages REFINES Receiving
  ...
  WHERE
    ⊕ : q ∈ dom(pcut)
    ⊕ : m ∈ mark_m
  THEN
    ⊕ : mark_m := mark_m \ {m}
```

```
EVENT ReceivingBeforeCut REFINES Receiving
  ...
  WHERE
    ...
    ⊕ : q ∉ dom(pcut)
    ⊕ : m ∉ mark_m
    ...
```

– ReceivingPostCutMessages: 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. The message (m) received by the process (q) is removed from the set ($mark\_m$) of marked *post-snapshot* messages.
– ReceivingBeforeCut: 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. The actions of this event are the same as the actions of the *"normal"* event Receiving.

6.3. Deriving Asynchronous Snapshot Algorithms

The Lai and Yang Algorithm The Lai and Yang algorithm [16] 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 ASYNC-PROCESS, an abstract model of the Lai and Yang algorithm [16]: 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 LAI-PROCESS, refining ASYNC-PROCESS, localizes informations and describes a model for the Lai and Yang algorithm. We can identify, in the LAI-PROCESS model, events representing the phases (A) and (B) of the Lai and Yang algorithm:

EVENT StartingSnapshot REFINES StartingSnapshot
  ANY
    $nsm, ncstate, nm$
  WHERE
    $grd1 : nsm \subseteq C \wedge nm \in C \nrightarrow \mathbb{P}(M) \wedge ncstate \in C \nrightarrow \mathbb{P}(M)$
    $grd2 : initiator \notin dom(pcut)$
    $grd3 : ncstate = cstate \cup \{d \mapsto \varnothing | d \in C \wedge prj2(d) = initiator\}$
    $grd4 : nsm = send\_mark \cup \{d | d \in C \wedge prj1(d) = initiator\}$
    $grd5 : nm = \{d \mapsto \varnothing | d \in nsm\}$
  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.).

The Chandy and Lamport Algorithm The Chandy and Lamport algorithm [13] 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 ASYNC-PROCESS, thanks to refinement. However, the machine FIFO-PROCESS and context FIFO-NETWORK (extension of the context NETWORK) introduce 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.

6.4. Comparing the Formal Modelling of Snapshot Algorithms

We have studied two categories of snapshot algorithms: synchronous snapshot algorithms and asynchronous snapshot algorithms.

In this section, we compare these two types of snapshot algorithms according to two criteria: first, we make comparisons of formal development complexity; then, we compare their local and global characteristics.

Model	Total	Auto		Interactive	
NETWORK	6	6	100%	0	0%
FIFO-NETWORK	5	4	80%	1	20%
SYSTEM	55	50	90.9%	5	9.1%
OBSERVATION	41	37	90.24%	4	9.76%
SYNC-PROCESS	45	41	91.11%	4	8.89%
ASYNC-PROCESS	96	66	68.75%	30	31.25%
LAI-PROCESS	85	46	54.12%	39	45.88%
FIFO-PROCESS	229	12	5.24%	217	94.76%
LOC-FIFO-PROCESS	5	4	80%	1	20%
TOTAL	567	266	46.91%	301	53.09%

Table 2. Summary of Proof Obligations

Asynchronous snapshot algorithms seem at first glance more complex than synchronous ones: we can see in the previous sections that asynchronous algorithms possess more steps than synchronous algorithms. Asynchronous snapshot algorithms have two phases: 1) a snapshot initialisation phase (event StartingSnapshot) and 2) a progression phase (event ProgressingSnapshot); whereas, synchronous algorithms only have one phase: the simultaneous computation of the snapshot by the processes (event ProcessingSnapshot). Clearly, this difference affects the size of the models: obviously, formal models of asynchronous snapshot algorithms are more complex, since they contain more events. The difference in complexity is underlined by the number of invariants needed to demonstrate the consistency of a snapshot: the ASYNC-PROCESS model requires a set of invariants (A, B, C, see Fig.3), while the SYNC-PROCESS model only necessitates one (A, see Fig.2). Furthermore, the table 2, presenting the proof obligations discharged either manually or automatically, confirms that modelling asynchronous algorithms is more complex: the total number of proofs (96) for these algorithms is more than twice as much as the number of proofs (45) for synchronous algorithms, and it should be noted that the number of manual proofs for asynchronous algorithms is far greater (30 vs 4). The following table sums up the complexity differences between synchronous and asynchronous snapshot algorithms.

Complexity	**Synchronous**	**Asynchronous**
Invariants (consistency of snapshot)	*less* (1)	*more* (3)
Number of Algorithmic Steps	*less* (1)	*more* (2)
Number of Events related To The Computation	*less* (1)	*more* (2)
Number of Proofs	*less* (45)	*more* (96)

Table 3. Complexity of the Models of Snapshot Algorithms

If we analyse their local and global characteristics, we can see that synchronous snapshot algorithms are generally based on the observation of external global elements, such as global time, while, on the other hand asynchronous snapshot algorithms only relies on the local resources and informations of the processes. The table 4 summarises these characteristics:

Characteristics	**Synchronous**	**Asynchronous**
Global	*yes*	*no*
Local	*yes*	*yes*

Table 4. Local and Global Characteristics of the Models of Snapshot Algorithms

In a nutshell, we say that the availability of global shared elements (e.g. global time) greatly simplifies the formal design and modelling of distributed snapshot algorithms [30]. However this simplification of design is gained at the expense of other qualities, like localisation.

7. 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 provides an abstract view of 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 and can be refined following the same methodology preserving correctness. 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 models called ASYNC-PROCESS and SYNC-PROCESS, which express the underlying computation procedure 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 communications, namely channels and messages: the $mark$ variable is either a marker for a bit, a marker for *fifo* channels or a marker for global temporal aspects. The complexity of the development is measured by the number of proof obligations which are automatically/manually discharged (see table 2). The main difficulty of the development was the expression of a consistent snapshot in the machines ASYNC-PROCESS and SYNC-PROCESS, therefore the establishment of the refinement relation between these machines and the machine OBSERVATION. A set of invariants (A,B,C) of the machine ASYNC-PROCESS (Fig.3) and the invariant (A) of the machine ASYNC-PROCESS (Fig.2) were the keys of the development, where the generated proof obligations were quite difficult to discharge. We also notice that the number of manual proof obligations increases dramatically for the model FIFO-PROCESS (see table 3): this augmentation is due to the transformation of sets of messages into FIFO queues of messages. In fact, we had to prove a lot of properties defining the proper behaviours of FIFO queues and communications. 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 [24], our work provides an architecture for developing the snapshot algorithm using essential safety properties together with a formal proof that asserts its correctness.

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. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.R., Cansell, D., Méry, D.: A mechanically proved and incremental development of ieee 1394 tree identify protocol. Formal Asp. Comput. 14(3), 215–227 (2003)
3. Awadhesh Kumar Singh, P.K.J.: Message efficient global snapshot recording using a self stabilizing spanning tree in a manet. International Journal of Communication Networks and Information Security (IJCNIS) 3(3) (December 2011)
4. Back, R.J., Kurki-Suonio, R.: Distributed cooperation with action systems. ACM Trans. Program. Lang. Syst. 10(4), 513–554 (1988)
5. Back, R.J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. Distributed Computing 3(2), 73–87 (1989)
6. Back, R.J., Sere, K.: Stepwise refinement of action systems. Structured Programming 12(1), 17–30 (1991)
7. Bjorner, D., Henson, M.C. (eds.): Logics of Specification Languages. EATCS Textbook in Computer Science, Springer (2007)

8. Cansell, D., Méry, D.: The Event-B Modelling Method - Concepts and Case Studies. In: Bjoerner, D., Henson, M. (eds.) Logics of Specification Languages, pp. 33–140. Monographs in Theoretical Computer Science, Springer (Feb 2008), http://hal.inria.fr/inria-00579550
9. Cansell, D., Méry, D.: Designing old and new distributed algorithms by replaying an incremental proof-based development. In: Abrial, J.R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. Lecture Notes in Computer Science, vol. 5115, pp. 17–32. Springer (2009)
10. Chalopin, J., Métivier, Y., Morsellino, T.: On snapshots and stable properties detection in anonymous fully distributed systems (extended abstract). In: Even, G., Halldórsson, M. (eds.) Structural Information and Communication Complexity, Lecture Notes in Computer Science, vol. 7355, pp. 207–218. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31104-8_18
11. Chandy, K.M., Misra, J.: Parallel Program Design A Foundation. Addison-Wesley Publishing Company (1988), iSBN 0-201-05866-9
12. Chandy, K.M., Go, B., Mitra, S., White, J.: Towards verified distributed software through refinement of formal archetypes. In: IFIP Working Conference on Verified Software: Workshop on Experiments (October 2008)
13. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
14. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA$^+$ proof system: Building a heterogeneous verification platform. In: Cavalcanti, A., Déharbe, D., Gaudel, M.C., Woodcock, J. (eds.) International Conference on Theoretical Aspects of Computing - ICTAC 2010. Lecture Notes in Computer Science, vol. 6255, p. 44. Springer, Brazil Natal (2010), the original publication is available at www.springerlink.com
15. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (december 1999), iSBN 0-262-03270-8
16. Lai, T.H., Yang, T.H.: On distributed snapshots. Inf. Process. Lett. 25(3), 153–158 (1987)
17. Lamport, L.: Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
18. Leavens, G.T., Abrial, J.R., Batory, D.S., Butler, M.J., Coglio, A., Fisler, K., Hehner, E.C.R., Jones, C.B., Miller, D., Jones, S.L.P., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) GPCE. pp. 221–236. ACM (2006)
19. Massingill, B.L., Chandy, K.M.: Parallel program archetypes. In: Proceedings of the Scalable Parallel Library Conference. pp. 1–9 (1997)
20. Méry, D., Mosbah, M., Tounsi, M.: Refinement-based verification of local synchronization algorithms. In: Butler, M., Schulte, W. (eds.) FM. Lecture Notes in Computer Science, vol. 6664, pp. 338–352. Springer (2011)
21. Méry, D., Singh, N.K.: Analysis of dsr protocol in event-b. In: Défago, X., Petit, F., Villain, V. (eds.) SSS. Lecture Notes in Computer Science, vol. 6976, pp. 401–415. Springer (2011)
22. Mohamed, O.A., Muñoz, C., Tahar, S. (eds.): Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5170. Springer (2008)
23. Morgan, C.: Global and logical time in distributed algorithms. Inf. Process. Lett. 20(4), 189–194 (1985)
24. Ogata, K., Huyen, P.T.T.: Specification and model checking of the chandy & lamport distributed snapshot algorithm in rewriting logic. In: ICFEM 2012 (2012)
25. Owre, S., Shankar, N.: A brief overview of pvs. In: Mohamed et al. [22], pp. 22–27
26. Project RODIN: Rigorous open development environment for complex systems. http://www.eventb.org/ (2004-2010)

27. Szymanski, B., Shi, Y., Prywes, N.: Synchronized distributed termination. IEEE Transactions on Software Engineering 11(10), 1136–1140 (1985)
28. Tel, G.: Introduction to Distributed Algorithms. Cambridge Unversity Press (1994)
29. Wenzel, M., Paulson, L.C., Nipkow, T.: The isabelle framework. In: Mohamed et al. [22], pp. 33–38
30. Yang, Z., Marsland, T.A.: Global snapshots for distributed debugging: An overview. Tech. rep., Computing Science Department, University of Alberta (1992)

Manamiary Bruno Andriamiarina is currently pursuing his PhD at the Laboratory LORIA at the Université de Lorraine in Nancy, France. His main area of research interest is in Formal Modelling, Refinement of Models, Event-Driven Development, Dynamic Distributed Systems and Algorithms.

Dominique Méry is professor in computer science at Université de Lorraine since 1993. He is the Head of the Formal Methods Department of the LORIA Laboratory, a joint structure between CNRS, INRIA and Université de Lorraine. He is developing researches on formal methods and applications in the MOSEL team. His topics are formal modelling and applications of formal methods to embedded systems, healthcare systems and distributed systems.

Neeraj Kumar Singh received the Doctor in Computer Science degree from Henri Poincaré University, Nancy 1 (now the University of Lorraine), France, in 2011. He received his MS degree in Optimization System and Security (OSS) from University of Technology of Troyes, France, in 2008. He also received Master of Computer Application (MCA) from Uttar Pradesh Technical University (UTPU), India, in 2006, and B.Sc in Computer Science in 2003. From January 2012 to August 2013, he was a research associate in the Computer Science Department of University of York, UK. In August 2013, he joined McMaster University, Canada, as a postdoctoral researcher in the Department of Computing and Software Information Technology. His current research interests are focused on software engineering, software and system certification, model driven engineering in the automotive domain, formal methods, biological environment modeling, cyber-physical systems, automatic code generation, concurrent systems, and formal verification of the medical protocols. The results of his research works are published in several articles. He is also the author of a book *"Using Event-B for Critical Device Software Systems"*.

Received: January 22, 2013; Accepted: November 17, 2013.