Formal Proofs of Termination Detection for Local Computations by Refinement-Based Compositions

Maha Boussabbeh[1,2](✉), Mohamed Tounsi[2], Mohamed Mosbah[1], and Ahmed Hadj Kacem[2]

[1] LaBRI Laboratory, University of Bordeaux, Talence, France
{maha.bousabbah,mohamed.mosbah}@labri.fr
[2] ReDCAD Laboratory, University of Sfax, Sfax, Tunisia
{ahmed.hadjkacem,mohamed.tounsi}@fsegs.rnu.tn

Abstract. In this paper, we propose a formal framework enhancing the termination detection property of distributed algorithms and reusing their specifications as well as their proofs. By relying on refinement and composition, we show that an algorithm specified with local termination detection, can be reused in order to compute the same algorithm with global termination detection. The main idea relies upon the development of distributed algorithms following a top/down approach and the integration of additional computation steps developed in a pre-defined module. This module is specified in a generic and scalable way in order to be composed with particular developments. Once the composition link is proven, the global termination emerges automatically.

Keywords: Distributed algorithms · Local computations · Termination detection · SSP algorithm · Composition · Event-B method

1 Introduction

1.1 Overview

It is widely agreed that implementing distributed systems poses major problems and remains a real challenge. Distributed termination detection is one of the most important problems in distributed computing [11]. It is closely related to many other problems such as determining a causally consistent global state [10], detecting deadlocks [12], etc. Contrary to sequential algorithms, the termination of distributed computing is neither simple nor clear: what does termination mean in distributed algorithms? Can processors be aware of global termination? Is it essential for a processor to distinguish between its termination and the termination of the entire computation? Different termination detection modes have been defined [8] to relate the local state of processors with the global state of the network. The two modes we are interested in are the following: *Local Termination Detection* (LTD), i.e., each processor is able to determine only its own termination condition; and *Global Termination Detection* (GTD), i.e., at least one processor knows when the entire computation has finished on the network. P. Castéran et al. [7] have proved that it is quite interesting for a processor to detect that the algorithm has globally terminated. However, if we don't have a global perspective and we are only interested in local interactions, it will not be evident to know if the distributed algorithm has finished.

In this paper, we propose a general framework, based on Event-B, for transforming correct algorithms with LTD into algorithms with GTD. We rely on the high level abstraction of local computations [20], and we focus on formal proofs of termination, using a refinement-based composition. The prime objective is to provide a proof-based development which can be reused for building and ensuring global termination. Another objective is to show the effectiveness of combining a *correct-by-construction* [19] approach with a compositional reasoning for preserving properties and reusing proofs.

1.2 Related Works

Formal specifications are often beneficial, and provide a real help for expressing correctness with respect to safety properties in the study of distributed computing. This paper is not an exception in this respect. Formal approaches have been proposed to deal with the correctness of such algorithms in different contexts: solving gathered problems [13], revisiting snapshot algorithms [3], etc. Numerous studies related to the termination detection problem have been done. However, no clear idea, about the way which would be better for transforming distributed algorithms from LTD to GTD mode and reusing their proofs, has been come out from these works. To the best of our knowledge, no reusable formal approach has been published yet, clarifying how far it can save efforts of designers and how far it can be reused in particular developments.

Related works have been proposed to suggest solutions for algorithms detecting only the LTD mode. Two major algorithms are used to cope with this problem and build the *GTD* mode: the Dijkstra-Scholten algorithm [14] and the algorithm by Szymanski, Shi and Prywes (the SSP algorithm for short) [22]. E. Godard et al. [16] proposed to compose two graph relabelling systems, one encoding a given algorithm and another encoding a termination detection algorithm such as the Dijkstra-Scholten algorithm or the SSP algorithm. They proved that the resulting relabelling system transforms the first algorithm from LTD mode to GTD mode. However, this transformation modifies the algorithm and makes the computation steps of the nodes more complex.

V. Filou et al. [15] proposed to compose formal specifications of distributed algorithms with formal specifications of the Dijkstra-Scholten algorithm: let A be a distributed algorithm specified with LTD mode. Based on the Event-B method, authors proved that a node being in a terminal state of A can execute the Dijkstra algorithm in order to detect the instant where every other node has computed its final value. The specification as well as the proofs of the first algorithm (algorithm A) are reused when detecting the global termination. However, a composition of the Dijkstra algorithm cannot be proposed as a general approach for dealing with algorithms encoded with LTD mode: this algorithm is based on the election process. Several conditions were found to allow election algorithms: P. Castéran et al. [7] and J. Chalopin et al. [9] characterized families of graphs that admit this algorithm.

1.3 Contribution

In a previous work [5], we proposed a proof based development for transforming a spanning tree algorithm with LTD mode into the same algorithm with GTD mode. We relied on Event-B refinement and we specified a combination of the SSP and the spanning tree algorithm following a top down approach. In this paper, we generalize our approach and we propose a formal framework enhancing the termination detection property of distributed algorithms without altering their specifications. Our framework is based on formal specifications and proofs of termination, encapsulated in a separate module according to the *SSP* algorithm. This module is developed in a generic and scalable way in order to be composed with particular developments.

Let A be a distributed algorithm where processors do not detect the global termination of A. Our main goal is to compose A with SSP and produce a correct algorithm which (i) reuses specifications as well as proofs associated to the algorithm A and (ii) enables processors to detect the global termination of the computation. We investigate necessary requirements for such a composition and give users guidelines for reusing the *SSP* algorithm in several developments with correctness.

The refinement of models is the key element, allowing preservation of correctness proofs. Moreover, using pre-defined modules, developed in a high level abstraction, makes our approach reusable. Based on the *correct-by-construction* approach and on the modularization [18] technique, we achieve our aim and show, with examples, what users gain with the proposed approach. In this paper, we illustrate our approach by the *3-colouring* of a ring specified with the *LC1* synchronisation. The main objective of this simple example is to demonstrate the use of our work during models development. Our approach is also applied to complex case studies such as the Mazurkiewicz [21] algorithm, specified with the *LC2* synchronization. The illustration of this algorithm gives us new results which do not appear in this paper, but they will be presented in a future publication.

1.4 Organization of the Paper

The paper is organized as follows: Sect. 2 recalls basic concepts of local computations and Event-B method. Section 3 presents the *SSP* algorithm and introduces our approach where we describe the composition process of *SSP* with Event-B developments. Section 4 details formal specifications and proofs of this composition. Section 5 illustrates our approach by an example. Finally, a short discussion, conclusion and ongoing work round the paper up.

2 Preliminaries

2.1 Local Computations Model

In this section, we illustrate, in an intuitive way, the notions of local computations, and particularly those of graph relabelling systems by showing how some algorithms on networks of processors may be encoded within this framework [20]. As usual, such a network is represented by a graph whose vertices (nodes) stand for processors and edges for (bidirectional) links between processors. Each vertex represents an entity that is capable of performing computation steps, sending and receiving messages. We consider anonymous networks with asynchronous message passing, i.e., each computation may take an unpredictable, but finite, amount of time. At every time, each vertex and each edge are in some particular state which will be encoded by a vertex or an edge label. According to its own state and to the states of its neighbours, each vertex may decide to do an elementary computation step. After this step, the state of this vertex, its neighbours and the corresponding edges may be changed according to some specific computation rules. Moreover, it is supposed that once a node reaches a final state it remains in such a state until the end of the algorithm.

The graph relabelling systems meet the following requirements: (i): they do not change the underlying graph, but they change only the labelling of their components (edges and/or vertices). The final labelling is the result. (ii) they are local, that is, each rewriting changes only a connected subgraph of a fixed size in the underlying graph. A sub-graph contains a subset of the vertices and edges in a graph G. (iii) they are locally generated, that is, the applicability condition of the rewriting depends only on the local context of the relabelled sub-graph.

The distributed aspect comes from the fact that several rewriting steps can be performed simultaneously on "far enough" subgraphs, giving the same result as a sequential realization of them, in any order. A large family of classical distributed algorithms encoded by graph rewriting systems is given in [20].

2.2 Event-B

The Event-B [1] modelling language defines mathematical structures as contexts and formal model of the system as machines. The context is defined by abstract sets, constants, and axioms which describe properties of constants. An Event-B machine describes a reactive system, using a set of invariant properties and a finite list of events modifying state variables. Recently the Event B language and its tool support Rodin [2] have been extended with the possibility to define a *module interface* i.e., logical unit containing *callable* operations. The important characteristic of these modules is that they can be developed separately and, when needed, incorporated and instantiated in the main system development. According to A. Iliasov et al. [18], a *module interface* is a separate Event-B component specifying a set of services. It encapsulates external variables, constants, invariants, and a collection of operations characterised by their *pre-/post-conditions*.

When a *module interface M* is *imported* into an Event-B machine (via the clause *USES*), an instance is created. Several instances can be created for the same module. To avoid name clashes, each instance is added with a prefix *pre* chosen by the user. Consequently, all the names *imported* from the module appear with the corresponding prefix. The importing machine can invoke the operations by means of events and read the external variables of *M*. As presented below, an interface operation *Op* is characterized by its *pre-* and *post-conditions* [18]. The *pre-conditions* contain a list of predicates applied on parameters *par* and on external variables v to define the states when an operation may be invoked. The primed variables v' and res', defined in the operation *post-condition* (*post*), stand for the final variable values after the operation execution. If some primed variables are not mentioned, the corresponding variables are unchanged by the operation.

The execution of a called operation is abstractly modelled by an Event-B event, named *calling_Op*. Internal parameters (*in_par*) are evaluated and passed to the operation *Op*. In addition to calculating a result (*result*), an operation call can also update the external variables (v). A set of proof obligations are generated to guarantee that the state of the module is protected by the operations.

```
OPERATION  Op  ≙
any     par
pre     pre(par, v)
return  result
post    post(par, v', res')
end
```

$\sqsubseteq$

```
Event  calling_Op  ≙
any    in_par
where  guard(in_par, pre_v)
then   resultOp := pre_Op(in_par)
end
```

3 SSP Composition with *Correct-by-Construction* Developments

3.1 The SSP Algorithm

We consider a distributed algorithm which terminates when all nodes reach their local termination conditions. The *SSP* algorithm [22] detects an instant in which the entire computation is achieved in the network. Let G be a graph. Each node $n0$ is associated with a predicate $P(n0)$ and an integer $a(n0)$. $P(n0)$ depends on the local termination of $n0$. Once a node $n0$ detects its local termination, the value of $P(n0)$ can be transformed from $FALSE$ to $TRUE$. $a(n0)$ is introduced to specify the fact that all nodes being in a distance equal to $a(n0)$ have locally terminated. Initially $P(n0)$ is $FALSE$ and $a(n0)$ is equal to -1. Transformations of the value of $a(n0)$ are defined by the following rules.

– $if\ P(n0) = FALSE\ Then\ a(n0) = \ -1,$
– $if\ P(n0) = TRUE\ Then\ a(n0) = 1 + min\{a(n_k) | k \geq 0\ and\ k \leq d\}.$

Let $n0$ be a node and let $\{n_1, ..., n_d\}$ be the set of nodes adjacent to $n0$; d stands for the number of edges the node $n0$ has to other nodes, called the degree of $n0$. The new value of a$(n0)$ depends on values associated with $n0$ and its neighbours.

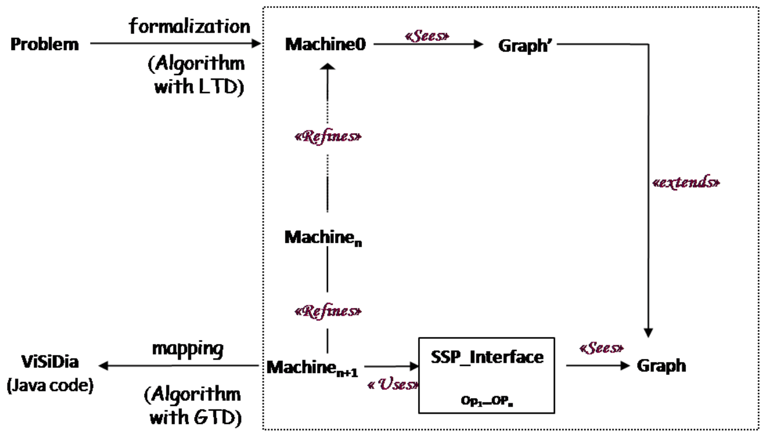

Fig. 1. SSP composition with Event-B development

3.2 Proposed Approach

We summarize in this section the idea of the SSP composition with a distributed algorithm expressed in Event-B. Our aim is to prove that for any algorithm satisfying LTD, expressed in event-B and proved independently, the algorithm obtained by the composition with SSP satisfies the same specification with GTD. Note that an Event-B development is based on a *correct-by-construction* approach [19] which supports an incremental process controlled by the refinement of models. In previous works [15,23], authors proposed Event-B patterns containing proof-based guidelines and describing how a distributed algorithm can be correctly designed: the development can start with a very abstract model, then, by successive refinements, we obtain a concrete one that expresses the local behaviour (state) of processors in the network. As presented in Fig. 1, we consider a problem, which can be formalized through a distributed algorithm with *LTD*, developed by a chain of Event-B machines. We suppose that the size of the graph is known by all the nodes. Let A be the distributed algorithm. The formalization process is based on the refinement of models, using the RODIN [2] platform.

- The chain of refinement $Machine_0...Machine_n$ expresses the *Problem* in an incremental development.
- The $Machine_n$ defines local interactions between nodes according to the computation process of the algorithm A.
- The refinement of $Machine_n$ by $Machine_{n+1}$ produces a set of events corresponding to some additional computation steps to detect the global termination of the algorithm A. These computations are specified according to the *SSP* rules.
- The *SSP_Interface* is a predefined logical entity that we have developed and proved independently. More precisely, it is an Event-B Interface containing

callable operations, corresponding to the execution process of the *SSP* rules. Moreover, it contains general proofs of termination detection for local computations.

- The $Machine_{n+1}$ *uses* the *SSP_Interface*, means that it can have access to all the proofs discharged in the SSP_Interfce and execute its operations.
- The *Graph* is an Event-B context that we have developed to specify basic properties of a network which represents the application field of distributed algorithms. Basically, a network is defined as a connected, simple and undirected graph. The *Graph'* context *extends Graph*, means that it can use all the specifications defined in the *Graph* context and introduce other static properties, describing the particularity of the algorithm A.
- A translation of Event-B specifications into a java code can be generated by relying on *B2Visidia* tool [24]. Thus, a solution can be mapped from the $Machine_{n+1}$ into *ViSiDiA* [4], i.e., *ViSiDiA* is a platform for simulating, visualizing and testing local computations. Note that this item is not studied in this paper.

4 Formal Descriptions

4.1 Network Specification: The *Graph* Context

The *Graph* context describes basic properties of the network on which distributed algorithms are running. Formally, a network can be straightforwardly modelled as a connected, undirected and simple graph where nodes denote processors and edges denote point-to-point communication links. An undirected graph means that there is no distinction between the two nodes associated with each edge (see axm4). A graph is simple if it has at most one edge between any two nodes (see axm2 and axm3) and no edge starts and ends at the same node (see axm5). A graph (directed or not) is connected if, for each pair of nodes, a path joining these two nodes exists(see axm6). According to D. Cansell et al. [6], a connected graph g over a set of finite nodes ND (see axm1) can be presented as follows:

$$\boxed{\begin{array}{l} axm1 : finite(ND) \\ axm2 : g \subseteq ND \times ND \\ axm3 : dom(g) = ND \\ axm4 : g = g^{-1} \\ axm5 : ND \lhd id \cap g = \varnothing \\ axm6 : \forall s \cdot s \subseteq ND \land s \neq \varnothing \land g[s] \subseteq s \Rightarrow ND \subseteq s \end{array}}$$

4.2 The *SSP_Interface*

The *SSP_Interface* is an Event-B component, specifying a set of additional computation steps that can be associated to the nodes in order to build a global termination detection. We assume that the local termination of the algorithm that we want to compose is stable, i.e., a terminated processor will not again be woken up in the course of further computation. We mean by termination, the termination of the algorithm that we want to compose. This component encapsulates formal specifications of the *SSP* algorithm into callable operations, modifying a set of external variables. These variables are defined as follows:

```
Interface SSP_Interface
SEES Graph
Variables
Local_TD
counter
Global_TD
...
OPERATION Update_Termination(...)
...
OPERATION To_Global_Termination(...)
...
OPERATION Global_Termination(...)
...
OPERATION Diffusion(...)
```

- $Local\_TD$ characterizes the local termination detection of a node: $Local\_TD \in ND \rightarrow BOOL$. Let n be a node. $Local\_TD(n) = True$ means that n has computed its final value. Initially, there is no node detecting the local termination.
- $counter$ leads a node to detect the global termination: $counter \in ND \rightarrow \mathbb{P}(\mathbb{N} \times \mathbb{Z})$. If n is a node, $(i \mapsto j) \in counter(n)$ means that at the computation step i, all the nodes, being in a j distance from n, detected locally the termination. Initially $counter(n) = \{0 \mapsto -1\}$. After i computation steps, $counter(n) = \{0 \mapsto -1, 1 \mapsto 0, ..., i \mapsto j\}$. We store the entire history of counters in order to investigate invariants of the model and simplify proofs during the development. In this paper, the *last counter* of n means the value of j calculated at the last computation step i. Formally, it can be specified as follows: last counter(n)= max(ran(counter(n))).
- $Global\_TD$ characterizes the global termination state of a node: $Global\_TD \in ND \rightarrow BOOL$. $Global\_TD(n) = True$ means that n is aware of the termination of the algorithm. Initially, there is no node detecting the global termination.

***UpdateTermination* Operation**. A node being in a terminal state, updates its $Local\_TD$ and acts on its $counter$. Let $counter_i(n)$ be the *last counter* computed by a node n. The new value of the $counter$ of n is defined at the computation step $(i + 1)$. This computation is specified by the *Update_Termination* operation. The pre-conditions (*Pre*) stand for the requirements of the operation execution. The terminal state of a node depends naturally on the algorithm that we want to compose. Thus the condition of the updating action is defined while calling this operation (see the example in Sect. 5). Note that the execution of this operation should be blocked once a node updates its $Local\_TD$ from *FALSE* to *TRUE*. This can be achieved by strengthening its pre-conditions by $(Local\_TD(n) = FALSE)$ and $(counter_i(n) = -1)$. The primed variables $Local\_TD'(n)$ and $counter'(n)$ defined in the post-conditions stand for the new values of $Local\_TD(n)$ and $counter(n)$ after the operation execution. In our case, we are interested in updating external variables without returning particular results.

`pre-conditions:` $Local\_TD(n) = FALSE$ and $counter_i(n) = -1$
`post-conditions:` $Local\_TD'(n) = TRUE$ and $counter'_{i+1}(n) = 0$

```
OPERATION  $Update\_Termination \;\widehat{=}$
any  $n, i$
pre  $pre1 : Local\_TD(n) = FALSE$
     $pre2 : i = max(dom(counter(n)))$
     $pre3 : max(ran(counter(n))) = -1$
return  $result1$
post  $post1 : Local\_TD'(n) = TRUE$
      $post2 : counter'(n) = counter(n) \cup \{(i+1) \mapsto 0\}$
      $post3 : result1' = TRUE$
```

***ToGlobalTermination* Operation**. Once a node n updates its *Local_TD* state, it computes a new value of the $counter$. Let i be the last computation step in which n has modified its $counter$. The new value of the $counter$ of n is defined at the computation step $(i+1)$, and depends on the *last counter* computed by the neighbours of n. Let $Ng(n)$ be the set of nodes adjacent to n. $Ng(n) = \{ng_1, ..., ng_d\}$. Let $\{counter_j(ng_1), ..., counter_k(ng_d)\}$ be the set of the *last counter* computed by each node in $Ng(n)$. Let C be the set of the *last counter* computed by each node in $(Ng(n) \cup \{n\})$. $C = \{counter_j(ng_1), ..., counter_k(ng_d), counter_i(n)\}$. $counter_{i+1}(n) = 1 + min(C)$. This computation step is specified by the operation *To_Global_Termination.* The execution of this operation should be blocked if the *counter* of the node n reaches the size of the graph (S), or if one of the nodes in $(Ng(n) \cup n)$ detects the global termination.

`pre-conditions:` $Local\_TD(n) = TRUE$ and $counter_i(n) < S$
n has no neighbour detecting the global termination
`post-conditions:` $counter'_{i+1}(n) = 1 + min(C)$

```
OPERATION  $To\_Global\_Termination \;\widehat{=}$
any  $n, i, Ng, C$
pre  $pre1 : Local\_TD(n) = TRUE$
     $pre2 : max(ran(counter(n))) < card(ND)$
     $pre3 : i = max(dom(counter(n)))$
     $pre4 : Ng = \{vi \cdot vi \mapsto n \in g | vi\}$
     $pre5 : C = \{a1, n1 \cdot n1 \in Ng \cup \{n\} \wedge a1 = max(ran(counter(n1))) | a1\}$
     $pre6 : \forall vi \cdot vi \in Ng \cup \{n\} \Rightarrow Global\_TD(vi) = FALSE$
return  $result2$
post  $post1 : counter'(n) = counter(n) \cup \{(i+1) \mapsto (1 + min(C)\}$
      $post2 : result2' = TRUE$
```

***GlobalTermination* Operation.** Within a finite number of steps, the *counter* of a node n can reach the size of the graph S, i.e., all the nodes being in a distance $\leq S$ from n have locally terminated. Thus, the node n can detect the fact that the entire computation is achieved in the network. Hence, it can update its *Global_TD* state and diffuse this information to its neighbours to make them aware of this global termination. The updating action of *Global_TD* state is specified by the operation *Global_Termination.* The execution of this operation is blocked once the node updates its *Global_TD* from *FALSE* to *TRUE.* This can be achieved by strengthening its pre-condition by $(Global\_TD(n) = FALSE)$.

`pre-conditions:` $counter_i(n) = S$ and $Global\_TD(n) = FALSE$
`post-conditions:` $Global\_TD'(n) = TRUE$

```
OPERATION  $Global\_Termination$  $\widehat{=}$
any  $n$
pre  $pre1 : max(ran(counter(n))) = card(ND)$
     $pre2 : Global\_TD(n) = FALSE$
return  $result3$
post  $post1 : Global\_TD'(n) = TRUE$
      $post2 : result3' = TRUE$
```

Diffusion **Operation.** Once a node n is aware of the termination of the entire computation in the network, it can transmit this information to its neighbours by updating the value of their *Global_TD* states. Each neighbour can in turn transmit the same information. This computation step is specified by the operation *Diffusion*. The execution of this operation is blocked when all the neighbours of the node n detect the global termination. This can be achieved by strengthening its pre-condition via a predicate, showing that n has at least one neighbour that is not detecting the global termination.

pre-conditions:	$Global\_TD(n) = TRUE$
	n has at least one neighbour v where $Global\_TD(v) = FALSE$
post-conditions:	n updates the $Global\_TD'$ values of its neighbours.

```
OPERATION  $Diffusion$  $\widehat{=}$
any  $n, Ng$
pre  $pre1 : Global\_TD(n) = TRUE$
     $pre2 : Ng = \{v \cdot v \in g[\{n\}] | v\}$
     $pre3 : \exists v \cdot v \in Ng \wedge Global\_TD(v) = FALSE$
return  $result4$
post  $post1 : Global\_TD' = Global\_TD \Leftarrow \{v \cdot v \in g[\{n\}] | v \mapsto TRUE\}$
      $post2 : result4' = TRUE$
```

4.3 Formal Proofs

The intention behind our approach is to compose distributed algorithms with the *SSP* specifications in order to build the global termination detection. Such a composition is achieved via calling the previous operations. In this section, we prove that the execution of these operations ensures the correctness of the resulting algorithm. More precisely, we investigate the invariants of the model and prove the following properties.

(P1): the resulting algorithm preserves the *LTD* property of the initial algorithm (Theorem 1). This property can be easily proved. In fact, we can prove that the *counter* computed by a node n increases during the different computation steps (Invariant 1). Thus, once a node computes a positive *counter*, the new value of this *counter* remains positive.

Invariant 1. $\forall n, i, a, i', a' \cdot i' < i \wedge (i \mapsto a) \in counter(n) \wedge i' \mapsto a' \in counter(n) \Rightarrow a' < a$

Invariant 2. $\forall n \cdot Global\_TD(n) = TRUE \Rightarrow (max(ran(counter(n))) \geq 0)$

Moreover, we can prove that the last counter $((max(ran(counter(n))))$, computed by a node n detecting the global termination, is a positive value (Invariant 2). Furthermore, once a node detects locally the termination, it sets the new value of its *counter*, through the execution of the *Update_Termination* operation, to zero. Thus, we can prove that a node n detects the local termination ($Local\_TD(n) = TRUE$) if and only if the value of the last *counter* calculated by this node is positive (Invariant 3).

Invariant 3. $\forall n \cdot Local\_TD(n) = TRUE \Leftrightarrow (max(ran(counter(n))) \geq 0)$

Theorem 1. $\forall n \cdot Global\_TD(n) = TRUE \Rightarrow Local\_TD(n) = TRUE$

(P2): If a node n updates its global termination state ($Global\_TD(n) = True$), every node v on the network has locally terminated (Theorem 2): once a node updates its *Local_TD* state, it increments the value of its *counter* by executing the operation *To_Global_Termination*. The new value of the *counter* depends on the values associated to the neighbours. We can prove that the difference between the maximum and the minimum of the *last counter* computed by two neighbours does not exceed 1 (Invariant 4).

Invariant 4. $\forall n, v, a, b \cdot v \in g[\{n\}] \wedge a = max(ran(counter(n))) \wedge b = max(ran(counter(v))) \Rightarrow (max(\{a, b\}) - min(\{a, b\}) \leq 1)$

Let *chains* be the set of possible chains in the graph, i.e., connected edges, and $Nodes(ch)$ be the set of the nodes concerned in a chain ch.

$$\boxed{\begin{array}{l} axm7 : chains = \{x1, x2, t, nodes \cdot x1 \in ND \wedge x2 \in ND \wedge x1 \neq x2 \wedge nodes \subseteq ND \\ \qquad \wedge \{x1, x2\} \subseteq nodes \wedge t \in nodes \setminus \{x2\} \rightarrowtail\!\!\!\!\twoheadrightarrow nodes \setminus \{x1\} \wedge t \subseteq g \wedge (t \neq t^{-1}) | t\} \\ axm8 : Nodes \in chains \rightarrow \mathbb{P}(ND) \\ axm9 : \forall ch \cdot ch \in chains \Rightarrow Nodes(ch) = ran(ch) \cup dom(ch) \end{array}}$$

Let $E = \{n \cdot n \in Nodes(ch) | max(ran(counter(n)))\}$ be the set of the *last counter* computed by $Nodes(ch)$, and $card(ch)$ be the size of the chain ch. Note $(Max - Min)$ is the difference between the maximum and the minimum of E. We prove, by induction on the size of ch, that $(Max - Min \leq card(ch) - 1)$ (Invariant 5). Furthermore, the *last counter*, computed by the first node detecting the global termination, reaches the size of the graph S ($S = card(ND)$). Thus, the maximum of E is equal to S. Moreover, $card(ch) \leq S$. Hence, $Max - Min \leq S$. Therefore, $Min \geq 0$. Consequently, we prove that if a node detects the global termination, all the other nodes have locally terminated (Theorem 2).

Invariant 5. $\forall ch, N \cdot ch \in chains \wedge N = Nodes(ch) \Rightarrow max(\{n \cdot n \in N | max(ran(counter(n)))\}) - min(\{n \cdot n \in N | max(ran(counter(n)))\}) \leq card(N) - 1$

Theorem 2. $(\forall n \cdot Global\_TD(n) = TRUE) \Rightarrow (\forall v \cdot v \in ND \Rightarrow Local\_TD(v) = TRUE)$

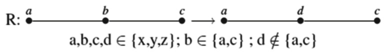

Fig. 2. 3-Colouring of a ring

5 Example: 3-Colouring of a Ring

Consider a ring with at least 3 nodes. The 3-colouring problem consists in assigning a color to each node from a set of three ones. Two neighbours have different colors. Let $\{x, y, z\}$ be the set of colors. The corresponding relabelling system is defined by considering the rule R (Fig. 2). A new context should be added to extend the *Graph* context in order to specify formal properties of a ring. Building a correct model may start with a very abstract machine and then, by successive refinements, we obtain a concrete one in which we specify the relabelling rule. We refine the last level by introducing a new machine in which we clarify the local termination of the nodes, and we use the *SSP_Interface* in order to build a global termination. According to the computation steps of this algorithm, we affirm that a node n reaches its final state when n and its neighbours get different colors. Hence, the following invariant should be added: assume that *Col* is a function introduced to characterize the color of the nodes.

Invariant 6. $\forall s \cdot s \in ND \Rightarrow (Col(s) \notin \{Col(g(s))\} \Leftrightarrow SSP\_Terminaison(s) = TRUE)$

The calling of the previous operations is similar to the previous example, except the execution of the *Update_Terminaton* operation which should be strengthened by a new guard (grd4), specifying when a node can update its termination state.

```
EVENT  $Calling\_Op1$
any $n, i$
where
$grd1 : SSP\_Local\_TD(n) = FALSE$
$grd2 : i = max(dom(SSP\_counter(n)))$
$grd3 : max(ran(SSP\_counter(n))) = -1$
grd4 : $(Col(n) \notin \{Col(g(n))\}$
then $act1 : res\_OP1 := SSP\_Update\_Termination(n \mapsto i)$
```

5.1 What We Gain

It seems that we have to do more work in order to compose an algorithm with the proposed specifications: we have to develop the algorithm in a progressive way controlled by the refinement of models. Then we add a new machine to specify a suitable implementation of the predefined operations. But we do have the following advantages:

– We don't need to prove the computation steps of the *SSP* algorithm. This is because we have already done this, when developing the *SSP_Interface*.

Table 1. Proof Statistics

Model	Total	Automatic		Interactive	
SSP_Interface	91	39	43 %	52	57 %
SpTree *M3* (with the *SSP_interface*)	44	31	70 %	13	30 %
SpTree *M3*(without the *SSP_interface*)	112	36	32 %	76	68 %
3-colouring *M3* (with the *SSP_interface*)	45	36	80 %	9	20 %
3-colouring *M3*(without the SSP_interface)	117	41	35 %	76	65 %

Thus, we have saved efforts of users on discharging proofs (see Table 1). The additional proof obligations, generated while introducing a new machine using the *SSP_Interface*, are not very complex to discharge: 70 % and 80 % of them are respectively proved automatically for the two examples (the spanning tree as well as the 3-colouring of a ring). Note that we have tested building and proving the global termination of the two algorithms without using the proposed SSP_Interface. The new generated machines for the spanning tree and the 3-colouring algorithm produce respectively only 32 % and 35 % of proof obligations discharged automatically.

– We reuse all the proofs associated to the first algorithm (the Spanning tree and the 3-colouring of a ring in our cases). The incremental proof-based process of refinement provides a way to preserve the correctness of the algorithm and to validate the integration of new requirements.
– We can reuse the proposed specifications in other case studies. The SSP_Interface is defined in a high level abstraction in order to build the global termination detection of distributed algorithms.

6 Discussion, Conclusion and Future Work

In this paper, we have proposed a proof-based framework for composing distributed computing with the *SSP* algorithm in order to build a global termination detection. The main characteristic of our approach is that it transforms algorithms from LTD to GTD mode, enables reuse in development, and saves efforts on proving distributed computing: by relying on the *SSP* algorithm, we specified a generic module containing formal specifications and proofs for the global termination detection. This module can be composed with simple and complex cases studies with different synchronizations. During the development, a list of proof obligations is generated by the Rodin [2] platform to ensure the safety of the development. We believe that this work has a number of benefits. In a nutshell, we say that composing a *correct-by-construction* development with the *SSP* algorithm enhances the termination detection property of distributed computing. Moreover, specifying *SSP* in a pre-defined module greatly simplifies the reuse of specifications and proofs.

It is worth noticing that if we look carefully at what the global termination means in distributed algorithms, we have to distinguish between the termination of the computing and the transmitting messages between processors. In this work, the global termination, we are interested in, is to detect the instant when all processors have computed their final values, i.e., no processor can modify its state. We don't detect the instant when there is no message in transit in the network. Moreover, our approach might be improved if we used the diameter of the graph instead of its size. But we made this choice for the sake of simplicity. As a future work, it would be interesting to take into account these limitations and study the case that local termination of processors is not stable. Moreover, added to safety properties, we think that it would be more relevant to ensure liveness properties [17]: when all nodes have locally terminated, the algorithm will eventually detect global termination. Furthermore, we aim to study and detail the last item presented in Sect. 3.2. Thus, we can implement *Java* codes of the proposed framework and simulate algorithms into the *Visidia* [4] platform. Starting with previous studies [24], we can translate our formal specifications into Java codes and propose a certified tool for transforming automatically distributed algorithms from LTD to GTD.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, New York (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-b. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Andriamiarina, M.B., Méry, D., Singh, N.K.: Revisiting snapshot algorithms by refinement-based techniques. Comput. Sci. Inf. Syst. **11**(1), 251–270 (2014)
4. Bauderon, M., Mosbah, M.: A unified framework for designing, implementing and visualizing distributed algorithms. Electr. Notes Theor. Comput. Sci. **72**(3), 13–24 (2003). http://dx.doi.org/10.1016/S1571-0661(04)80608-X
5. Boussabbeh, M., Tounsi, M., Hadjkacem, A., Mosbah, M.: Towards a general framework for ensuring and reusing proofs of termination detection in distributed computing. In: 24rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion Crete, Greece, 17th-19th February 2016 (2016)
6. Cansell, D., Méry, D.: The event-B modelling method: concepts and case studies. In: Bjørner, D., Henson, M.C. (eds.) Logics of Specification Languages. Monographs in Theoretical Computer Science, pp. 47–152. Springer, Berlin (2008)
7. Castéran, P., Filou, V.: Tasks, types and tactics for local computation systems. Stud. Inform. Univ. **9**(1), 39–86 (2011)
8. Chalopin, J., Godard, E., Métivier, Y.: Local terminations and distributed computability in anonymous networks. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 47–62. Springer, Heidelberg (2008)
9. Chalopin, J., Godard, E., Métivier, Y.: Election in partially anonymous networks with arbitrary knowledge in message passing systems. Distrib. Comput. **25**(4), 297–311 (2012). http://dx.doi.org/10.1007/s00446-012-0163-y

10. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985). http://doi.acm.org/10.1145/214451.214456
11. Chandy, K.M., Misra, J.: Parallel program design - a foundation. Addison-Wesley, UK (1989)
12. Chandy, K.M., Misra, J., Haas, L.M.: Distributed deadlock detection. ACM Trans. Comput. Syst. **1**(2), 144–156 (1983). http://doi.acm.org/10.1145/357360.357365
13. Courtieu, P., Rieg, L., Tixeuil, S., Urbain, X.: A certified universal gathering algorithm for oblivious mobile robots. CoRR abs/1506.01603 (2015)
14. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. **11**(1), 1–4 (1980). http://dx.doi.org/10.1016/0020-0190(80)90021-6
15. Filou, V., Mosbah, M., Tounsi, M.: Towards proved distributed algorithms through refinement, composition and local computations. In: 2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, 17–20 June 2013, pp. 353–358 (2013). http://dx.doi.org/10.1109/WETICE.2013.67
16. Godard, E., Métivier, Y., Mosbah, M., Sellami, A.: Termination detection of distributed algorithms by graph relabelling systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 106–119. Springer, Heidelberg (2002)
17. Hoang, T.S., Abrial, J.-R.: Reasoning about liveness properties in event-b. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 456–471. Springer, Heidelberg (2011)
18. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in event B development: modularisation approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010)
19. Leavens, G.T., Abrial, J., 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: Proceedings of 5th International Conference of Generative Programming and Component Engineering GPCE 2006, Portland, Oregon, USA, 22–26 October 2006, pp. 221–236 (2006). http://doi.acm.org/10.1145/1173706.1173740
20. Litovsky, I., Métivier, Y., Sopena, E.: Graph relabelling systems and distributed algorithms. In: Handbook of Graph Grammars and Computing by Graph Transformation, pp. 1–56. World Scientific Publishing Co., Inc., River Edge (1999)
21. Mazurkiewicz, A.W.: Distributed enumeration. Inf. Process. Lett. **61**(5), 233–239 (1997)
22. Szymanski, B.K., Shi, Y., Prywes, N.S.: Terminating iterative solution of simultaneous equations in distributed message passing systems. In: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, 5–7 August 1985, pp. 287–292 (1985). http://doi.acm.org/10.1145/323596.323623
23. Tounsi, M., Mosbah, M., Méry, D.: Proving distributed algorithms by combining refinement and local computations. ECEASST 35 (2010) http://journal.ub.tu-berlin.de/eceasst/article/view/442
24. Tounsi, M., Mosbah, M., Méry, D.: From event-b specifications to programs for distributed algorithms. In: 2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, 17–20 June 2013. pp. 104–109 (2013). http://dx.doi.org/10.1109/WETICE.2013.44