Electronic Communications of the EASST
Volume 35 (2010)


# Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

## Proving Distributed Algorithms by Combining Refinement and Local Computations

Mohamed Tounsi Mohamed Mosbah Dominique Méry

21 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Proving Distributed Algorithms by Combining Refinement and Local Computations

**Mohamed Tounsi**[2] **Mohamed Mosbah**[2] **Dominique Méry**[1]

Loria, Université Henri Poincaré Nancy 1 France[1]
LaBRI, Université Bordeaux 1 Talence France[2]

**Abstract:** Distributed algorithms are considered to be very complex to design and to prove; our paper contributes to the design of *correct-by-construction* distributed algorithms. The main idea relies upon the development of *distributed algorithms* following a top/down approach, which is clearly well known in earlier works of Dijkstra, and to use refinement for proving the correctness of the resulting algorithms. However, the link between the problem and the first model remains to be expressed and the refinement is a real help to justify in a very progressive way the choices of design. We propose in this work a framework combining local computations models and refinement to prove the correctness of a large class of distributed algorithms. Local computations models define abstract computing processes for solving problems by distributed algorithms and can be integrated into a the Event B modelling language to define proof-based patterns for the design of distributed algorithms. We illustrate our approach by examples like the leader election protocol or the distributed coloring algorithm. Our proposal is integrated into a environment called ViSiDiA.

**Keywords:** Proof-based pattern, Event-B, Local computation, Distributed algorithm.

## 1 Introduction

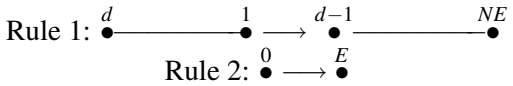
### 1.1 Overview

The *correct-by-construction* approach can be supported by a progressive and incremental process controlled by the refinement of models for distributed algorithms. Event-B modelling language is supporting our methodological proposal suggesting proof-based guidelines. The main objective is to facilitate the correct-by-construction approach for designing distributed algorithms [Reh09, BM09, Mér09] by combining local computing models [CM10, CM07c] and Event-B models [Abr10a] to get benefits of both models. In fact, local computation models provide an abstraction of distributed computations, which can be expressed in Event-B; they provide a graphical complement for Event-B models and Event-B models provide a framework for expressing correctness with respect to safety properties. More precisely, we introduce several methodological steps identified during the development of case studies. A general structure characterizes the relationship between the contract, the Event-B models, and the developed algorithm using a specific application of Event-B models and refinement. Distributed algorithms are considered with respect to the local computation model [CGM08] based on a relabelling relation over graphs representing distributed systems. The ViSiDiA toolbox [Mos09] provides facilities for simulating local computation models which can be easily modelled using Event-B with refinement.

### 1.2 The local computation model for deriving correct-by-construction distributed algorithms

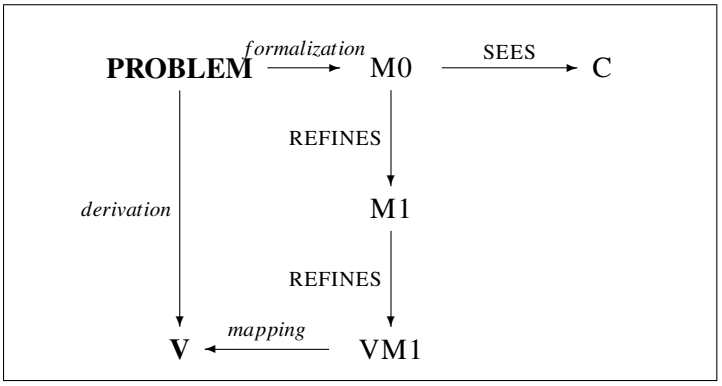
ViSiDiA [Mos09] is an environment for implementing, simulating, testing and visualizing distributed algorithms. It is based on the use of graph relabelling systems to encode distributed algo-

rithms and to prove their correctness. A distributed algorithm in the local computation model [CM10, CM07c] is simply given by some (possibly infinite but always recursive) set of rules, like for instance, the leader election:




A run of the algorithm consists in applying the relabelling rules specified by the algorithm until no rule is applicable, which terminates the execution. The relabelling rules are applied asynchronously and nondeterministically, which means that given the initial labelling usually many different runs are possible. The distributed aspect means that two consecutive non-overlapping steps may be applied in any order and in particular in parallel. The local computation model is easily expressed in the Event-B framework. The methodology described by the diagram, leads to define the following features from the problem analysis into a ViSiDiA solution:

- A context *C* states properties of graphs and the mathematical problem to solve (election, spanning tree, ...)
- A machine *M0* expresses the specification of the problem to solve by events stating a relation between the initial states and the final states; for instance, the leader election is stated by an event which states effectively the *one shot* election.
- The refinement of *M0* into a model *M1* states that the machine *M1* expresses the inductive property allowing to express the computation in the local computation model.
- The next refinement of *M1* (called *VM1*) is a refinement for producing a set of events corresponding to the set of relabelling rules.
- *V* is derived from *VM1* by a translation of *VM1* into ViSiDiA [Mos09].




## 1.3 Patterns for proof-based developments

Patterns and design patterns [GHJ+94] provide a very convenient help in the design of object-oriented software. Originally, they were borrowed from architectures practices and recently, J.-R. Abrial [Abr10b] suggested the introduction of a kind of patterns for the proof-based development. The action/reaction patterns have been applied to the press case study by J.-R. Abrial and they improve the proving process. Another pattern called re-usability pattern, has been suggested by Abrial, Cansell and Méry [AH07, CM07b]. Clearly, a growing activity on modeling patterns has started and is addressing many kinds of case studies and domains of problems. No classification is yet given and there is no real repository of patterns validated for a specific modeling language

based on proof-based transformations. There are current works [HFA09] on developing plugins to implement and to help the use of these structures.

In the *ViSiDiA* [Mos09] research project, we are interested in formal specification and formal proof of local computation systems. This activity consists in giving a formal semantics to these systems, in developing tools for the certification of such algorithms : proof of invariants, proof of termination, and also in comparing the computational power of various subclasses of local computation systems or other kinds of distributed computation paradigms. In this paper, we combine this high level encoding of distributed algorithms with Event-B modeling to describe a pattern for proving distributed algorithms. This pattern gives a way to derive easily a *ViSiDiA* system using built-in libraries and to construct a chain of models ensuring the correct by construction paradigm. *correct by construction*.

### 1.4 Organization of the paper

Section 2 describes basic concepts of the Event-B modeling language. Section 3 presents the local computations framework. In particularly, we concentrate on the graph relabeling systems which can be defined as a framework to encode a distributed system. Section 4 introduces our pattern for combining local computations models and we formally develop the different contexts, as well as the different machines, of the pattern. Section 5 applies our pattern to develop two problems of distributed algorithmics. Finally, we conclude this paper by summarizing our ideas and by describing future directions.

## 2 Modeling by Step-wise Refinement

Modeling software-based systems [Bac79, Abr10a, CM07a] is based on transition systems, which are stating how the state of a system is modified, when actions (or transitions or events) are operating on the state. Each action (or transition or event) is characterized by a list of state variables, a condition over these variables and an effect or a transformation of the current state into a next state. State variables $x$ and the current value of the variable $x$ is simply expressed by the symbol $x$, whereas the next value of the variable $x$ is expressed by the symbol $x'$. An action (or an event $e$) over state variables is defined by a *before-after* predicate denoted $BA\_e(x,x')$ stating the modification of the variable $x$ with respect to $e$. A model (or a machine) $M$ is defined by a context defining mathematical structures (sets, constants, properties of constants), a list of (state) variables $x$, a predicate $I(x)$ stating invariant properties satisfied by the state variable $x$, a predicate defining the possible initial values of state variables and a finite set of events $e_1,\ldots,e_n$. State variables can be modified using a general form denoted by the construct $x :| P(x,x')$. This should be read: $x$ *is modified in such a way that the predicate* $P(x,x')$ *holds*, where $x'$ denotes the *new value* of the vector and $x$ denotes its *old value*. A general form of event can be used by adding a parameter $t$ and a guard $g(t,x)$ to produce the following event form: $e \mathrel{\widehat{=}}$ **when** $G(x)$ **then** $x :|Q(x,x')$ **end**) or ($e \mathrel{\widehat{=}}$ **any** $t$ **where** $G(t,x)$ **then** $x :|R(x,x',t)$ **end**). The EVENT-B modeling language [Abr10a] is providing such structures for defining mathematical structures into *contexts* and formal model of system into *machines*. An EVENT-B model is internally correct, when *proof obligations* proves that events preserve the invariant $I(x)$ and that each event is feasible, i.e. when the event $e$ is enabled, next values exist. Proof obligations are generated by the RODIN tool [Pro04] and can be discharged either automatically by an integrated proof tool or through interactive proof steps. A EVENT-B model describes a reactive system characterized by a finite list of events modifying a state variable; an operational interpretation of a EVENT-B model states that traces of the current model can be generated from the initial states and applying events. Moreover, the ProB tool [LB08] provides a tool-set for animating and for simulating finite instances of EVENT-B models

by setting sets, constants and parameters. These functionalities helps in validation of EVENT-B models; however, they require to use finite models (finite sets, ...), contrary to proof obligations which are proved for any model using the proof tool on proof obligations. The price to pay is to accept that a part of proof obligations may remain unproved after the use of the automatic proof procedures of the RODIN platform and lead the user to interact with the proof tool to discharge remaining unproved proof obligations. Unproved proof obligations should be carefully analyzed, since an unproved proof obligation can be false and the user should modify the model and starts a new proof process. However, an unproved proof obligation may indicate that a modification should be made on the current model.

Since models may generate very *complex* proof obligations, the development of proved models can be improved by the refinement process. The key idea is to combine models and elements of requirements using the refinement. The refinement [Bac79, Bac98] of a machine allows us to enrich a model in a *step-by-step* approach, and is the foundation of our *correct-by-construction* approach. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is done by extending the list of state variables, by refining each abstract event into a corresponding concrete version, and by adding new events. In Fig.1, the diagram at the left illustrates the refinement-based relationship among events and models.
We suppose that an abstract model $AM$ with variables $x$ and invariant $I(x)$ is refined by a concrete model $CM$ with variables $y$. The abstract state variables, $x$, and the concrete ones, $y$, are linked together by means of the, so-called, *gluing invariant* $J(x,y)$. A number of proof obligations ensure that (1) each abstract event of $AM$ is correctly refined by its corresponding concrete version of $CM$, (2) each new event of $CM$ refines *skip*, which is intending to model *hidden* actions over variables appearing in the refinement model $CM$. More formally, if $BA\_ae(x,x')$ and $BA\_ce(y,y')$ are respectively the abstract and concrete before-after predicates of events, we say that $ce$ in $CM$ refines $ae$ in $AM$ or that $ce$ simulates $ae$, if one proves the following statement corresponding to proof obligation: $I(x) \wedge J(x,y) \wedge BA\_ce(y,y') \Rightarrow \exists x' \cdot (BA\_ae(x,x') \wedge J(x',y'))$. To summarize, refinement guarantees that the set of traces of the abstract model $AM$ contains (modulo stuttering) the traces of the concrete model $CM$.

In an EVENT-B model (called either machine or refinement), the clause INVARIANT and the clause THEOREMS list safety properties satisfied by the model, when proof obligations are discharged; however, the clause INVARIANT should be an inductive property for the list of events and the clause THEOREMS is a logical consequence of the clause INVARIANT. When one defines the clause INVARIANT, the main difficulty is to add enough informations to obtain an inductive property to discharge all required proof obligations. The incremental proof-based process of refinement through the chain of machines provides a way to diffuse the complexity of proofs through the machines and to validate integration of requirements. In Fig.1, the diagram at the right summarizes links between contexts ($CC$ extends $AC$; $AC$ defines the set-theoretical logical and problem-based theory of level $i$ called $\mathscr{T}h_i$, which is extended by the set-theoretical logical and problem-based theory of level $i$ called $\mathscr{T}h_{i+1}$, which is defined by $CC$). Each machine ($AM$, $CM$) sees set-theoretical and logical objects defined from the problem statement and located in the CONTEXTS models ($AC$, $CC$). The abstract model $AM$ of the level $i$ is refined by $CM$; state variables of $AM$ is $x$ and satisfies the invariant $I(x)$; the refinement of $AM$ by $CM$ is checking the invariance of $J(x,y)$ and does need to prove the invariance of $I(x)$, since it is obtained freely from the checking of $AM$.

The management of proof obligations is a technical task supported by the RODIN tool [Pro04], which provides an environment for developing correct-by-construction models for software-based systems according to the diagram at the right of Fig.1. Moreover, the RODIN platform integrates a tool for animating EVENT-B models and for model-checking finite configurations of EVENT-B

models at different steps of refinement.

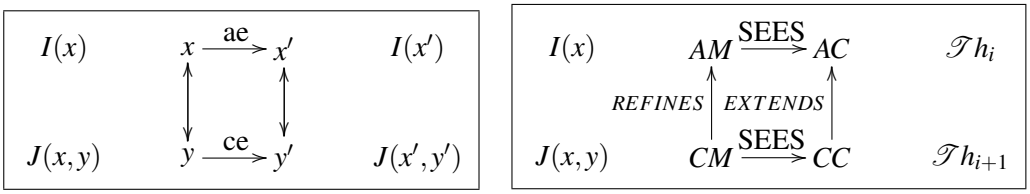



Figure 1: Machines and Contexts relationship

# 3 Local Computation Models

In this section, we illustrate, in an intuitive way, the notion of local computations, and particularly that of graph relabelling systems by showing how some algorithms on networks of processors may be encoded within this framework [LMS99]. As usual, such a network is represented by a graph whose vertices stand for processors and edges for (bidirectional) links between processors. At every time, each vertex and each edge is in some particular state and this state will be encoded by a vertex or edge label. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may have changed according to some specific *computation rules*. Let us recall that graph relabelling systems satisfy the following requirements:

(C1) they do not change the underlying graph but only the labelling of its components (edges and/or vertices), the final labelling being the result,

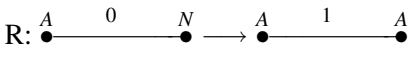
(C2) they are local, that is, each relabelling changes only a connected subgraph of a fixed size in the underlying graph,

(C3) they are locally generated, that is, the applicability condition of the relabelling only depends on the local context of the relabelled subgraph.

For such systems, the distributed aspect comes from the fact that several relabelling 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 relabelling systems is given in [BMMS02]. In order to make the definitions easy to read, we give in the following two examples of a graph relabelling system for computing a spanning tree and for coloring a ring. Then, the formal definitions of local computations will be presented.

## 3.1 Distributed computation of a spanning tree

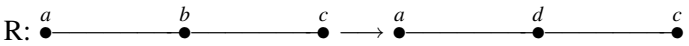
Let us first illustrate graph relabelling systems by considering a simple distributed algorithm which computes a spanning tree of a network. Assume that a unique given processor is in an "active" state (encoded by the label **A**), all other processors being in some "neutral" state (label **N**) and that all links are in some "passive" state (label **0**). The tree initially contains the unique active vertex. At any step of the computation, an active vertex may activate one of its neutral neighbours and mark the corresponding link which gets the new label **1**. This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the links with label **1**. An elementary step in this computation may be depicted as a *relabelling step* by means of the following relabelling rule $R$ which describes the corresponding label modifications (remember that labels describe processor status):



An application of this relabelling rule on a given graph (or network) consists in $(i)$ finding in the graph a subgraph isomorphic to the left-hand-side of the rule (this subgraph is called the *occurrence* of the rule) and $(ii)$ modifying its labels according to the right-hand-side of the rule.

### 3.2 3-coloring of a ring

Consider a ring with at least 3 nodes. The (vertex) 3-coloring problem consists in assigning to each node a color from a set of three colors such that two neighbours have different colors. In distributed computing, vertex coloring algorithms are mainly used for resource allocation. A vertex coloring defines a partial order on processors allowing them, for example, to execute their critical section according to the order defined by their respective colors. We provide a relabelling system to color a ring with 3 colors, starting from an arbitrary configuration. Let $\{x,y,z\}$ be the set of colors. Let $S_3$ be the relabelling system defined by considering the following rule R:




a,b,c,d ∈ {x,y,z}; b ∈ {a,c} ; d ∉ {a,c}

Initially, vertices are labelled at random. This relabelling system, defined by the previous rule, assigns a correct 3-coloring to the vertices of a ring.

### 3.3 Formal definition of local computations

Let us introduce a few notations. We consider graphs which are finite, undirected and connected without multiple edges and self-loops. If $G$ is a graph, $V(G)$ denotes the set of vertices and $E(G)$ denotes the set of edges. The distance between two vertices $u,v$, denoted by $d(u,v)$, is the length of a shortest path between $u$ and $v$. For a vertex $v$ and a positive integer $k$; the *ball* of radius $k$ with center $v$, denoted by $B_G(v,k)$, is the subgraph of $G$ induced by the set of vertices $V' = \{v' \in V \mid d(v,v') \leq k\}$. Let $L$ be an alphabet. A graph labelled over $L$ will be denoted by $(G,\lambda)$, where $\lambda : V(G) \cup E(G) \to L$ is the function labelling vertices and edges. The graph $G$ is called the underlying graph, and the mapping $\lambda$ is a labelling of $G$. Let $\mathscr{G}_L$ be the class of graphs labelled over some fixed alphabet $L$. Local computations are characterized by applications of rules such that: an application of a rule to a ball depends exclusively on the labels appearing in the ball and changes only these labels. The previous examples can be described by the following general model.

**Definition 1** A graph rewriting relation is a binary relation $\mathscr{R} \subseteq \mathscr{G}_L \times \mathscr{G}_L$ closed under isomorphism. The transitive closure of $\mathscr{R}$ is denoted $\mathscr{R}^*$.
An $\mathscr{R}-$rewriting chain is a sequence of labelled graphs $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_n$ such that for every $i$, $1 \leq i < n$, $\mathbf{G}_i \mathscr{R} \mathbf{G}_{i+1}$. A sequence of length 1 is called an $\mathscr{R}-$rewriting step (a step for short).

By "closed under isomorphism" we mean that if $\mathbf{G}_1 \simeq \mathbf{G}$ and $\mathbf{G} \mathscr{R} \mathbf{G}'$, then there exists a labelled graph $\mathbf{G}'_1$ such that $\mathbf{G}_1 \mathscr{R} \mathbf{G}'_1$ and $\mathbf{G}'_1 \simeq \mathbf{G}'$.

**Definition 2** Let $\mathscr{R} \subseteq \mathscr{G}_L \times \mathscr{G}_L$ be a graph rewriting relation.

1. $\mathscr{R}$ is a relabelling relation if whenever two labelled graphs are in relation then their underlying graphs are equal (not only isomorphic):

$$\mathbf{G} \mathscr{R} \mathbf{H} \quad \Longrightarrow \quad G = H.$$

When $\mathscr{R}$ is a relabelling relation we shall speak about $\mathscr{R}-$relabelling chains (resp. step) instead of $\mathscr{R}-$rewriting chains (resp. step).

2. A relabelling relation $\mathscr{R}$ is local if whenever $(G,\lambda)\ \mathscr{R}\ (G,\lambda')$, the labellings $\lambda$ and $\lambda'$ only differ on some ball of radius 1 :

$$\exists\, v \in V(G) \text{ such that } \forall\, x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x).$$

We say that the step changes labels in $B_G(v,1)$.

3. An $\mathscr{R}-$normal form of $\mathbf{G} \in \mathscr{G}_L$ is a labelled graph $\mathbf{G}'$ such that $\mathbf{G}\ \mathscr{R}^*\ \mathbf{G}'$, and $\mathbf{G}'\ \mathscr{R}\ \mathbf{G}''$ holds for no $\mathbf{G}''$ in $\mathscr{G}_L$. We say that $\mathscr{R}$ is noetherian if for every graph $\mathbf{G}$ in $\mathscr{G}_L$ there exists no infinite $\mathscr{R}-$relabelling chain starting from $\mathbf{G}$. Thus, if a relabelling relation $\mathscr{R}$ is noetherian, then every labelled graph has an $\mathscr{R}-$normal form.

The next definition states that a local relabelling relation is *locally generated* if its restriction on centered balls of radius 1 determines its computation on any graph.

**Definition 3** Let $\mathscr{R}$ be a relabelling relation. Then $\mathscr{R}$ is locally generated if the following is satisfied: For any labelled graphs $(G,\lambda)$, $(G,\lambda')$, $(H,\eta)$, $(H,\eta')$ and any vertices $v \in V(G)$, $w \in V(H)$ such that the balls $B_G(v,1)$ and $B_H(w,1)$ are isomorphic via $\varphi : V(B_G(v,1)) \longrightarrow V(B_H(w,1))$ and $\varphi(v) = w$, the following three conditions

1. $\forall\, x \in V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \eta(\varphi(x))$ and $\lambda'(x) = \eta'(\varphi(x))$,
2. $\forall\, x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x)$,
3. $\forall\, x \notin V(B_H(w,1)) \cup E(B_H(w,1)),\ \eta(x) = \eta'(x)$,

imply that $(G,\lambda)\ \mathscr{R}\ (G,\lambda')$ if and only if $(H,\eta)\ \mathscr{R}\ (H,\eta')$.

Finally, local computations are the computations defined by a relation locally generated. The reader can find in [LMS99] detailed definitions, formal properties and many examples of local computations.

Let us also note that labels can be sets or sets of sets. In particular, it is possible to handle graphs described as labels. For example, the Mazurkiewicz [Maz97] universal graph reconstruction is a distributed enumeration algorithm which allows the reconstruction of an anonymous graph. The manipulated labels for such an algorithm are sets standing for graphs (see [BMMS02]).

# 4 A pattern for combining local computation models

## 4.1 The pattern presentation

The Event-B modeling method provides the framework for supporting our method for developing distributed algorithms expressed in the local computation model. In fact, this method can perfectly be used to construct a pattern for combining local computation models and Event-B language. A pattern is a terme borrowed to the software engineering community and intends to capture elements that can be replayed when developing a given class of problems. Our pattern is a set of guidelines expressed by transformations over Event B models.

Section presents a description of the pattern, and we explain how it is used to construct a correct distributed algorithm in the local computation model. We describe concepts for modeling state-based systems and we explain how models are defined in the Event-B. In Figure 2, the high-level structure of the distributed algorithm pattern is shown, followed by a very brief explanation of the different pattern levels:

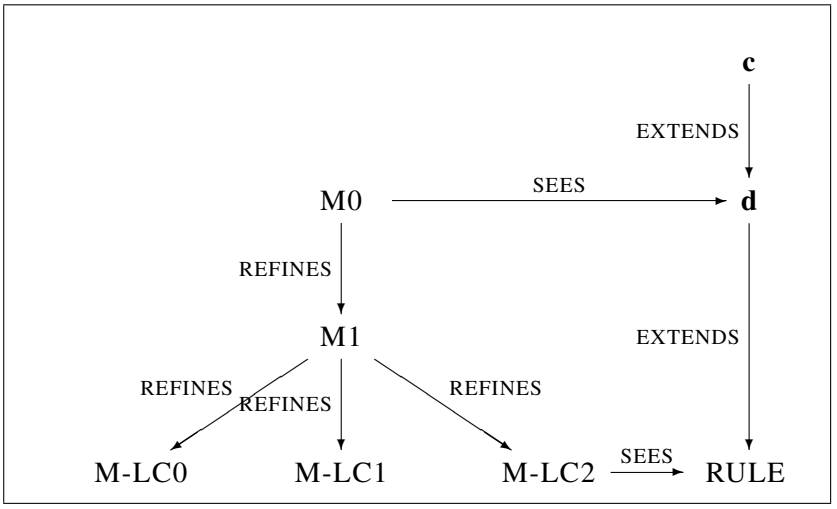



Figure 2: The distributed algorithm pattern

Generally, the design of a distributed algorithm starts with a very abstract model and then by successive refinement we obtain a concrete one that expresses the local behavior of the processor in the network. According to this development strategy and to our experience in the modelling of distributed algorithm, we conclude that three basic levels are necessary to build a correct model of a distributed algorithm with respect to a given problem to address (election, naming, spanning, ...). These levels are described as follows :

1. **M0** is an abstract machine modelling the result in *one shot*. It has only one event stating the result (*what*) in one shot and does not describe the algorithmic process (*how*) for computing the result.

2. The refinement of **M0** into **M1** will allow to express the inductive property which allows to explain how the computation is working in the local computation model. Observe that, **M1** level may be a set of machines used to express the working of the algorithm and describe the necessary steps to move from an initial state to a final state which corresponds exactly to the calculated result in the first stage.

3. Finally, the **M**-LC (**L**ocal **C**omputation **M**achine) represents the last machine of the specification. In this step, we introduce the rewriting rules of the algorithm.

With these machines, contexts are required with a particular definition in the specification. The first one is the "c" context, it defines the application field of the distributed algorithm: i.e: graph, tree, ring, ...The second one is the "d" context, it is defined as an extension of "c". It includes static properties that describe the particularity of each algorithm. The last one is the "rule" context, it defines all rewriting rules of the algorithm. However, this context has a virtual presence. The designer is not supposed to define it, but it will serve to verify the consistency of the "M-LCi" machine.

## 4.2 Formal development of the different contexts of the pattern

### 4.2.1 The "c" context

The "c" context describes the basic properties of the network on which distributed algorithms are running. Formally, a network can be straightforwardly modeled 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 two nodes associated with each edge (see axm3). A graph is simple, if has less than one edge between any two nodes and no edge starts and ends at the same node; it is obviouslye expressed by the choice of the representation of relation by a set. A graph (directed or not) is connected, if for each pair of nodes, there exists a path joining these two nodes (see axm5). According to Jean-Raymond Abrial et al. [CM07a], a connected graph namely "**g**" is modeled by a set of nodes namely "**ND**" can be presented as follows:

$$
\begin{array}{l}
axm1 : g \subseteq ND \times ND \\
axm2 : dom(g) = ND \\
axm3 : g = g^{-1} \\
axm4 : id(ND) \cap g = \emptyset \\
axm5 : \forall s.s \subseteq ND \wedge s \neq \emptyset \wedge g[s] \subseteq s \Rightarrow ND \subseteq s
\end{array}
$$

### 4.2.2 The "d" context

In the present section, we provide a formal definition of the "d" context which gives the specification of the algorithm properties. These properties are considered with respect to the computation model based on the graph relabeling systems. In such systems, each node and each edge is in some particular state which is encoded by a specific label [MO04]. This label allows nodes to perform an elementary step of computation according to some relabeling rules. Formally, we specify labels as a finite set "LN" for **L**abel **N**odes and "LE" for **L**abel **E**dges. In order to specify correctly the rewriting system, it is necessary to mention that only nodes should change their states. In other words "LN" must contain at least two different labels, but "LE" can include one or more than one label. When the algorithm begins execution, each node [resp. edge] starts with a specific label. This initial labeling is very important since it influences on the apply of the algorithm rewriting rules. Let "init_LE" (initial edge labels) and "init_LN" (initial node labels) be two sets to encode the graph initial labeling.

According to [LMS99], if we only consider a noetherian graph relabeling systems (which means that from any initial graph, no infinite relabeling chain exists) any computation on a graph eventually gives a result in a finite time. This means that after a finite computation steps and when no rule can be applied, the execution of the algorithm is finished and its solution is computed.

$$
\begin{array}{l}
axm1 : finite(LN) \wedge finite(LE) \\
axm2 : card(LN) > 1 \wedge card(LE) \geq 1 \\
axm3 : final\_LN \subseteq LN \wedge final\_LE \subseteq LE \\
axm4 : init\_LN \subseteq LN \wedge init\_LE \subseteq LE \\
axm5 : final\_LN \neq \emptyset \wedge final\_LE \neq \emptyset \\
axm6 : init\_LN \neq \emptyset \wedge init\_LE \neq \emptyset \\
axm7 : init\_LN \cup final\_LN = LN \\
axm8 : init\_LE \cup final\_LE = LE \\
axm9 : solution \subseteq (g \rightarrow final\_LE) \times (ND \rightarrow final\_LN) \\
axm10 : solution \neq \emptyset
\end{array}
$$

Formally, we define "solution" as a non-empty set of possible and reachable solutions by the algorithm on the graph (when the graph becomes irreducible which means that, no rule can be applied). It encodes a particular representation of the graph when nodes and edges are in a final

state. We denote final state of nodes [resp. edges] by a set called “final_LN” [resp. “final_LE”] representing all node label [resp. edge] when the algorithm execution terminates.

### 4.2.3 The “rule” context

A relabeling rule can be represented as a relabeling relation on the graph. In other words, it can be considered as a “before/after” relation that describes the change of states. Formally, we represent it as a binary relation called “rules” that could be shared by all synchronization algorithms. The “rules” is specified as follow: rules $\in$ (LN $\times$ $\mathbb{P}$(LN $\times$ LE) ) $\nrightarrow$ (LN $\times$ $\mathbb{P}$(LN$\times$ LE)). The left-hand side of “rules” represents the necessary condition to apply the rule. The first “LN” in the condition represents the state of the node center, where the powerset (LN $\times$ LE) represents the state of one (or more) neighbors as well as the state of edges connecting them to the center. The right-hand side of “rules” represents new labels of nodes and edges involved in the rule (result of the rule application).
In order to ensure the consistency of “rule” specification, we added some requirement properties: “rules” is a not empty set (axm3), a rule must perform a real label change (axm4) and the neighbors labels is a finite set (axm5). Formally, these properties are presented as follows:

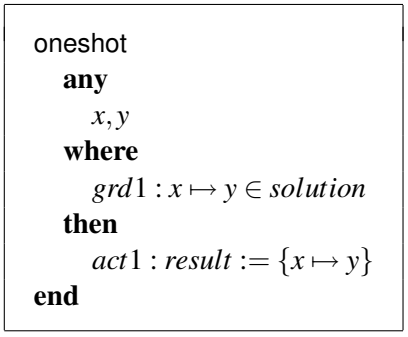
$$
\begin{array}{l}
axm3 : rules \neq \emptyset \\
axm4 : \forall cond, act \cdot cond \in (LN \times \mathbb{P}(LN \times LE)) \wedge cond \mapsto act \in rules \Rightarrow cond \neq act \\
axm5 : \forall v1, v2, c1, c2 \cdot c1 \in LN \wedge c2 \in LN \wedge v1 \in \mathbb{P}(LN \times LE) \wedge \\
v2 \in \mathbb{P}(LN \times LE) \wedge (c1 \mapsto v1) \mapsto (c2 \mapsto v2) \in rules \Rightarrow finite(v1) \wedge finite(v2)
\end{array}
$$

## 4.3 Formal development of the different machines of the pattern

After describing the basic mathematical structure, we can now proceed to the development of the distributed algorithm models. As we have previously explained, building a distributed algorithm specification consists of developing gradually a specification in three main levels:

### 4.3.1 The first level

In this very abstract level, the machine will express only the goal of the distributed algorithm and does not describe the process of computing the solution. As we presented in [CM07a], the election algorithm (IEEE 1394 protocol) can be done in one step. The specification of the algorithm is made by a machine having only one event for selection of a node called leader in a finite time. We used the same approach to develop spanning tree algorithms in [THMM09]. We have specified these algorithms with a machine that contains only one event which computes the solution in one shot and generates a spanning tree in the graph. In a similar manner, the “MAZURKIEWICZ” [AR08] algorithm has been developed by adopting the same approach. In order to specify this event, we introduce the variable “result” which will contain the result of the algorithm execution. Formally, “result” is defined by this very simple invariant : result $\in$ ( g $\leftrightarrow$ LE ) $\leftrightarrow$ ( ND $\leftrightarrow$ LN ). The first machine is the generic solution for all distributed algorithms problems. Effectively, we consider it as a basis of refinement to generate a specific algorithm models. It includes an event called “oneshot” which avows the result of the distributed algorithm when its execution is completed. In other words, there is no protocol, only the formal



definition of its intended result. The analogy of someone closing and opening their eyes. Formally, the "oneshot" event attributes an element from "solution" set to the "result" variable.
In order to avoid such difficulty in proofs we have chosen to declare the solution as a pair (x,y) instead of a single variable and therefore the generated PO will be proved using only the interactive prover of Rodin tool.

### 4.3.2 The second level

In this level, we introduce machine(s) as well as event(s) to allow computation on the graph. As mentioned earlier, this computation does not change the underlying graph but only the labeling of its components (nodes/edges). This level remains in a high level abstraction, it encodes the algorithm and computes its result without considering relabeling rules (they will be introduced in the last level). We add two functions "ln" and "le" that assign a label to each node and to each edge. The addition of these two variables involve the addition of new properties in the invariant component. The specification of these functions will take the following forms: ln $\in$ ND $\rightarrow$ LN and le $\in$ g $\rightarrow$ LE. The two next events establishes the invariants; the oneshot event is refined by modifying the guard and the action of the abstract event.

INITIALISATION
**begin**
$act1 : le :\in g \rightarrow init\_LE$
$act2 : ln :\in ND \rightarrow init\_LN$
**end**

oneshot
**refines** oneshot
**when**
$grd1 : le \mapsto ln \in solution$
**witnesses**
$x : x = le$
$y : y = ln$
**then**
$act1 : result := \{le \mapsto ln\}$
**end**

Remember that, a formal development with Event-B method of a distributed system is a sequence of models, linked by a refinement topology which is based on some methodological reasoning. Here, we observe that this reasoning is difficult to be systematic and until now it cannot be developed automatically. Because, each algorithm has its specificity, and with all formal methods (Event-B for instance), designer is always invited to reason on an abstract representation of the system. Therefore, finding all the system's invariants, and the suitable induction of the refinement is considered as the most complex task in the model building. Thus, refinement and the choice of the right abstraction make more tractable proof obligations, generated automatically from the text of B models and make us sure that the invariant is in fact an inductive property. The introduction of this level in our pattern, will help the designer to bridge this gap.
Considered as a small model, our pattern will guide designer to develop the necessary models of his own specification. As a generic event, we propose to define "Computing+" that will specify the computation on an unknown number of nodes of the graph. It specifies the goal traced by all models appearing in this stage and that can be summarized in one word: the incursion to a final state.

The declared parameters in this event are :

- "new_state_nodes" [resp. "new_state_edges"] is defined to model new states of nodes [resp. edges] belonging to "nodes" set [resp. "edges"] if a computation step takes place.

- "nodes" is the set of nodes that will change its states.

- "edges" is the set of edges connecting nodes of "nodes".
- "new_label_nodes" [resp. "new_label_edges"] is defined as a set including all future label of "nodes" [resp. "edges"].

The guards of "Computing+" event specification is given as follows:

$$
\begin{array}{l}
grd1 : le \mapsto ln \notin solution \\
grd2 : nodes \subseteq ND \\
grd3 : edges \subseteq g \\
grd4 : \exists a \cdot a \in nodes \wedge a \in ln^{-1}[init\_LN] \\
grd5 : \exists at \cdot at \in edges \wedge at \in le^{-1}[init\_LE] \\
grd6 : (\forall a \cdot a \in nodes \Rightarrow (\exists b \cdot b \in nodes \wedge a \mapsto b \in g)) \wedge (edges \subseteq nodes \lhd g \rhd nodes) \\
grd7 : new\_label\_nodes \subseteq final\_LN \\
grd8 : new\_label\_edges \subseteq final\_LE \\
grd9 : new\_state\_nodes \in nodes \nrightarrow new\_label\_nodes \\
grd10 : new\_state\_edges \in edges \nrightarrow new\_label\_edges \\
grd11 : new\_state\_nodes \neq \emptyset
\end{array}
$$

In the invariant component, "grd1" states that the actual situation of the graph is different from "solution". "grd4" [resp. "grd5"] is introduced to ensure that at least one active node [resp. edge] must appear in "nodes" [resp. "edges"] set.
The first part of grd6 checks if elements of "nodes" are connected (a necessary condition to apply relabeling rules on "nodes") and the second part ensures that "edges" is a subset of the edges connecting the elements of "nodes".
Active nodes are defined as nodes that will change their states in a computation step. Whereas the neighbors of the active nodes which do not change their states but they are used to match the rewriting rule are called "passive" [CMZ04]. All the other nodes that are not participating in such elementary relabeling step are called "idle". From this definition, we can conclude that some nodes (or edges) don't change their state in a rewriting step. For that reason, we have defined "new_state_nodes" and "new_state_edges" (see grd9 and grd10) as two partial functions.
"grd11" ensures that the new state of "nodes" is different from their previous state. Finally, actions of the event will update labels of nodes [resp. edges] which belong to "nodes" [resp. "edges"].

$$
\begin{array}{l}
act1 : ln := ln \Leftarrow new\_state\_nodes \\
act2 : le := le \Leftarrow new\_state\_edges
\end{array}
$$

However, after a bibliographic study we shall be able to confirm that sometimes a backward step in the computation may be considered by the distributed algorithm (Spanning tree: sequential computation nodes with ID for example [THMM09]). In other words, if we have nodes in a final state, they can decide to change its label to a one of the previous states. This kind of computation may be specified by Event-B. To do this, we add an optional event called "Computing-" that will be very similar to "Computing+". The difference between them can be summarized as follows:

- grd4 : The active node has to be an element from "final_LN" set,
- grd5 : The active edge has to be an element from "final_LE" set,
- grd7 : "new_label_nodes" becomes a sub set of "init_LN" ( new_label_nodes $\subseteq$ init_LN )
- grd8 : "new_label_edges" becomes a sub set of "init_LE" ( new_label_edges $\subseteq$ init_LE).

### 4.3.3 The third level

Once machine(s) of the second level has (have) been specified and proven, the last machine can be refined for describing local label modification. Now, we are able to observe the specification more precisely. Therefore, we can see more events, namely relabeling rules of the distributed algorithm. In this level, the “oneshot” event remains unchanged and the “Computing+” (and “Computing-”) event still exists: it will be refined by another event(s) that simulates the elementary computation step of each node. Note that, the number of new events must always be equal to the number of algorithm rules.
Three kinds of local computations are considered for implementing distributed algorithms: LC0, LC1 and LC2. This section will show in more details a general model of event for each kind of synchronization.

#### (a) Local Computation of type 0 (LC0)

In this level, we refine the “calcul+” event to obtain a LC0 event which can only perform a computation on two adjacent nodes in the network. It can change labels of two neighboring nodes as well as the label of their common edge. In order to specify this event (we called “Local_computation”), we add more details and we parameterize all abstract variables of the “calcul+”. The abstract variables are parameterized by replacing them with concrete values by means of “witness”. In Event-B, witness is defined as a simple equality predicate involving the abstract parameters.
In the variable component of “Local_computation”, we declare “x” and “y” as two adjacent nodes involved in the computation step. Let “new_label_x” and “new_label_y” be the two labels that will encode the new states of “x” and “y” after applying the rule. Let “new_label_edge” be the new label of the edge joining “x” and “y”. We note that the adding of “new_label_y" and “new_label_edge" variables is optional.
As an example, the witness “nodes: nodes = {x , y}”, present in the witness component, replaces the abstract parameter ‘ ‘nodes”, declared in the left hand side of the predicate, with the tow nodes “x” and “y”. Below the complete list of all witnesses.

$$
\begin{array}{l}
edges : edges = \{x \mapsto y, y \mapsto x\} \\
nodes : nodes = \{x, y\} \\
new\_state\_nodes : new\_state\_nodes = \{x \mapsto new\_label\_x, y \mapsto new\_label\_y\} \\
new\_state\_edges : new\_state\_edges = \{(x \mapsto y) \mapsto new\_label\_edge\} \\
new\_label\_nodes : new\_label\_nodes = \{new\_label\_x, new\_label\_y\} \\
new\_label\_edges : new\_label\_edges = \{new\_label\_edge\}
\end{array}
$$

In the guard component, “grd8” verify the existence of a rule that can be triggered with the initial state of nodes and/or edge. The “grd2” allow checking if the two declared nodes are neighbors or not. Formally, guards of the “Local_computation” for the LC0 relabeling rule can be encoded as follows :

$$
\begin{array}{l}
grd1 : le \mapsto ln \notin solution \\
grd2 : x \mapsto y \in g \\
grd3 : new\_label\_x \in final\_LN \\
grd4 : new\_label\_y \in final\_LN \\
grd5 : new\_label\_edge \in final\_LE \\
grd6 : ln(x) \in init\_LN \\
grd7 : le(x \mapsto y) \in init\_LE \\
grd8 : (ln(x) \mapsto \{ln(y) \mapsto le(x \mapsto y)\}) \mapsto \\
(new\_label\_x \mapsto \{new\_label\_y \mapsto new\_label\_edge\}) \in rules
\end{array}
$$

**(b) Local Computation of type 1 (LC1)**

LC1 is the star rule that updates a single node label and occasionally, labels of edges outgoing from the center. In a computation step, the label attached to the center of the star is modified according to some rules depending on the labels of the star. However, leaves labels are never modified (leaves are all the neighbors of the center). Like LC0, the parameterizing of abstract variables in LC1 event is also expressed by predicates in the witness component. They allow to introduce these four variables :

- "c" : It represents by convention the center of the star. All the neighbors of "c" are represented by g[{c}].
- "new_label_c" : It is the label that will encode the new state of "c" after the rule take place.
- "new_label_edge" : It is a sub set of "final_LE". It represent all the new states of the star edges.
- "Sphere_edge" : It is defined as a partial function mapped from each edge in the star to a label from "new_label_edge". The goal of this variable is to hold the new edges state.

"c" and "new_label_c" are considered as essential variables to specify LC1 rules when the other variables are considered as optional. In the guard component, we add an axiom to verify the existence of at least one edge in initial state. Also, we add an axiom to guaranty the existence of a relabeling rule that can coincide exactly with the initial state of the star and allows the triggering of the event.

**(c) Local Computation of type 2 (LC2)**

LC2 is the star rule that updates labels attached to the center and/or the leaves, according to some relabeling rules. Compared to LC1 model, additional requirement must be enforced to specify a LC2 algorithm. This requirement is given below : The "new_label_c" is removed and it is replaced by "Sphere_node" to hold the new state of the star (leaves and center). "Sphere_node" is defined as a partial function mapped from each node of the star to a label from "new_label_node". The "new_label_node" is defined as a subset of "final_LN". The "Sphere_node" specification is done as follow:

$$
\begin{array}{l}
Sphere\_node \in (g[\{c\}] \cup \{c\}) \nrightarrow new\_label\_node \\
Sphere\_node \neq \emptyset
\end{array}
$$

The other difference with the LC1 algorithm resides in the action component. In fact, in the substitution component of a LC2 event, "ln" is overwritten by elements of "Sphere_node": $ln := ln \lhd Sphere\_node$.

# 5 Examples

This section presents an application of our pattern and demonstrates how it can be used. To this end, we choose two examples: the first one is the "spanning tree" algorithm implemented with the LC0 synchronization and the second one is the "Coloration of a Ring" algorithm implemented with the LC1 synchronization. Both algorithms are given in section 3.1. Through these examples, we list and we discuss the initialization of the different pattern levels to generate quickly a correct specification.

The usage of design patterns in Object Oriented technology results in adapting and incorporating some pre-defined pieces of codes in a software project [AH08]:

- The adaptation of an Event-B design pattern essentially consists of instantiating its constants, variables and events in order to have them corresponding to some elements of the problem at hand.
- The incorporation of an Event-B design pattern within a larger model whose construction is in progress, consists of composing the design pattern events within some existing events of the model, so that the resulting effect is a refinement of the large model.

In this section, we apply these two techniques on our pattern to create an instance of the chosen distributed algorithms and we illustrate how we obtain an instance by applying refinement techniques.

## 5.1 Spanning Tree algorithm

### 5.1.1 The “c” context instantiation

Having the definition of the current graph namely, “g” over the set of nodes namely, “ND” in the pattern, we will now extend the context “c” to define elements of the tree. A tree can be defined as a connected acyclic subgraph that contains all the nodes of the graph and some edges. In order to specify a tree, we have to define a root “r”, a node: r $\in$ ND and a parent function “t” (each node has an unique parent node, except the root). “r” is the root of “g”, if there exists a path joining “r” to each node of the graph “g”. (for more information about tree building, the reader should see [CM07a]). Finally, we introduce the constant “trees” to be the set of all spanning trees (with root r) of the graph “g” : $trees = \{t | t \in ND \setminus \{r\} \rightarrow ND \wedge (\forall q \cdot q \subseteq ND \wedge r \in q \wedge t^{-1}[q] \subseteq q \Rightarrow ND = q) \wedge t \subseteq g\}$

### 5.1.2 The “d” context instantiation

Considering the network representation “g” and the set of possible trees in “g” namely, “trees”, we now proceed to the definition of the visualization of the distributed algorithm. We implement labels describing all states of nodes and edges as defined previously in the relabeling system “R”. Now, we can instantiate “LN” and “LE” as two sets; the first includes “A” and “N” and the second includes “Marked” and “not_Marked” labels. We notice that for this algorithm, edge labels are important to determine a spanning tree, they mark edges belonging to the spanning tree. After that, we define “solution” as a particular representation of the graph when nodes and edges are in final state. In other words, “solution” constitutes the set of all combination of labeled graph that represent trees in “trees”. The definition of “solution” is given by “axm9”. The “d” context instancing extends at the same time the tow contexts: “d” and “c_instantiation”. Formally, all added properties are listed by the following axioms:

$$
\begin{array}{l}
axm1 : LN = \{A, N\} \\
axm2 : init\_LN = \{A, N\} \\
axm3 : final\_LN = \{A\} \\
axm4 : LE = \{Marked, not\_Marked\} \\
axm5 : init\_LE = \{not\_Marked\} \\
axm6 : final\_LE = \{Marked, not\_Marked\} \\
axm7 : A \neq N \\
axm8 : Marked \neq not\_Marked \\
axm9 : solution = \{sol, a \cdot a \in trees \wedge \\
sol = \{((a \times \{Marked\}) \cup ((g \setminus a) \times \{not\_Marked\}))\} \times \{ND \times \{A\}\} | sol\}
\end{array}
$$

### 5.1.3 The "rule" context instantiation

The current algorithm is encoded by the graph relabeling system which is based on only one rule "R". The "rule_instantiation" context is specified as an extension of the two contexts "rule" and "d_instantiation". In this context we define "Rules" as a formal specification of "R":

$$
\begin{array}{l}
axm1 : Rules \subset rules \\
axm2 : Rules = \{(A \mapsto \{N \mapsto not\_Marked\}) \mapsto (A \mapsto \{A \mapsto Marked\})\}
\end{array}
$$

### 5.1.4 The "m1" machine instantiation

The first machine remains unchangeable. In the second level, a new machine which refines "M1" will be added. This machine, called "m1_instance", uses the "d_instance" context and it defines an instance of "calcul+" event, which gradually computes the spanning tree in a progressive way. It labels some edges and some nodes and this will allow to add a new edge to the tree under construction. Let "x" and "y" be the two variables from "ND" and let x $\mapsto$ y be the edge linking "x" to "y". Let "new_state_nodes" and "new_state_edges" be the new labels of the nodes (x,y) and the edge linking them. The guards specification of "calcul+" of "m1_instance" is done as follow:

$$
\begin{array}{l}
grd1 : x \mapsto y \in g \\
grd2 : le \mapsto ln \notin solution \\
grd3 : \{x, y\} \subseteq ND \\
grd4 : ln(y) \in init\_LN \\
grd5 : le(x \mapsto y) \in init\_LE \\
grd6 : (\forall t, v \cdot t \mapsto v \in \{x \mapsto y, y \mapsto x\} \Rightarrow \\
(t \in \{x, y\} \wedge v \in \{x, y\})) \wedge (\forall a, b \cdot a \in \{x, y\} \wedge b \in \{x, y\} \wedge a \mapsto b \in g \Rightarrow a \mapsto b \in \{x \mapsto y, y \mapsto x\}) \\
grd7 : \{A\} \subseteq final\_LN \\
grd8 : \{Marked\} \subseteq final\_LE \\
grd9 : new\_state\_nodes \in \{x, y\} \nrightarrow \{A\} \\
grd10 : new\_state\_edges \in \{x \mapsto y, y \mapsto x\} \nrightarrow \{Marked\} \\
grd11 : new\_state\_nodes \neq \emptyset
\end{array}
$$

The following witnesses replace the abstract variables with the concrete one.

$$nodes : nodes = \{x, y\}$$
$$edges : edges = \{x \mapsto y, y \mapsto x\}$$
$$new\_label\_nodes : new\_label\_nodes = \{A\}$$
$$new\_label\_edges : new\_label\_edges = \{Marked\}$$

### 5.1.5 The "m2-LC0" machine instantiation

We will refer to the previous model to define a machine with respect to LC0 synchronization as presented in our pattern. This machine is an instantiation of the "m2-LC0" machine, it refines the instance of "m1" machine and it sees the "rule_instance" context. The last machine is given by the following specification.

Local_computation
**any**
$x, y$
**where**
$grd1 : le \mapsto ln \notin solution$
$grd2 : x \mapsto y \in g$
$grd3 : ln[\{x\}] = \{A\}$
$grd4 : le[\{x \mapsto y\}] = \{not\_Marked\}$
$grd5 : (ln(x) \mapsto \{ln(y) \mapsto le(x \mapsto y)\}) \mapsto (A \mapsto \{A \mapsto Marked\}) \in Rules$
$grd6 : ln[\{y\}] = \{N\}$
**witnesses**
$new\_label\_x : new\_label\_x = A$
$new\_label\_y : new\_label\_y = A$
$new\_label\_edge : new\_label\_edge = Marked$
**then**
$act2 : ln := ln \Leftarrow (\{x \mapsto A, y \mapsto A\})$
$act3 : le := le \Leftarrow \{(x \mapsto y) \mapsto Marked\}$
**end**

## 5.2 Coloration of a Ring

The present algorithm assigns a correct 3-coloring to nodes of a ring. In other words, the algorithm can ensure that no two successive nodes are colored the same. In this section, we give a formal description of the ring structure under which the algorithm is executed. It is done by an extension of the "'c" context. After that, we instantiate the "d" context by presenting the different labels that can be assigned to the nodes. Also, we give the set of the possibles solutions in the ring. Like the previous algorithm, the first level is given by a machine that include only the "oneshot" event. Due to the lack of space we give in the following only the last level.

### 5.2.1 The "c" context instantiation

A Ring is a set of linked nodes (ND) that may be represented graphically in circular or triangular form.
Formally, it can be defined as a non oriented cycle of edges joining all the nodes of the graph (axm11) where each node in the ring is related only to two neighbors. We define "r" as an oriented ring in the graph. Like the developpement of the tree structure (presented

$$axm10 : rg \subseteq ND \times ND$$
$$axm11 : rg = r \cup r^{-1}$$

above) we require proofs related to the closure of relation to define
the ring. Let “cr" be the transitive closure of “r” defined by the axioms axm3 to axm6. We prove by means of a theorem that, “r" is equal to “cr". In fact this theorem can answer to the question concerning the existence of a path between any two nodes in “r". However with this definition, a node may have more than two neighbors which contradicts the definition of a ring. For this, “r" is specified by means of a bijective function as presented in (axm7). However, the use of bijective function cannot ensure the definition of a single ring in the graph. For solving this problem, we add in the context the (axm8) to guaranty the uniqueness of "r”.

$$
\begin{array}{l}
axm1 : r \subseteq ND \times ND \\
axm2 : \forall s \cdot (s \subseteq ND \times ND \wedge ND \neq \emptyset \wedge s;r \subseteq s \Rightarrow ND \times ND \subseteq r) \\
axm3 : cr \subseteq ND \times ND \\
axm4 : r \subseteq cr \\
axm5 : cr;r \subseteq cr \\
axm6 : \forall u \cdot (u \subseteq ND \times ND \wedge r \subseteq u \wedge u;r \subseteq u \Rightarrow cr \subseteq u) \\
axm7 : r \in ND \rightarrowtail ND \\
axm8 : \forall s \cdot (s \subseteq ND \wedge s \neq \emptyset \wedge s \subseteq r^{-1}[ND] \Rightarrow ND \subseteq s) \\
axm9 : \forall i \cdot i \in ND \wedge i \in dom(r) \Rightarrow i \neq r(i)
\end{array}
$$

The fact that the ring “r" does not contain more separate circuits is formalized by saying that the ND set belongs to the non empty set “s" which belongs to ND and to the image of ND under the converse of “r".

### 5.2.2 The “d” context instantiation

In this context we proceed to the definition of the visualization of the “Coloration of a Ring" algorithm. In this algorithm, edges do not change their states. Hence, we implement only labels describing states of the nodes as defined previously. Let “COLORS" be the set of labels that encode the node states. It includes “BLUE", “WHITE" and “RED" labels and it is equal to “init_LN" and “final_LN" because, initially, nodes are labelled (colored) at random. After that, we define “solution” as a particular representation of the ring when nodes are in final state.

$$
\begin{array}{l}
axm1 : partition(COLORS, \{BLEU\}, \{WHITE\}, \{RED\}) \\
axm2 : solution = \{x, ncolor \cdot x \in ND \wedge ncolor \in ND \rightarrow COLORS \\
\wedge ncolor(x) \neq ncolor(r(x)) \wedge ncolor(x) \neq ncolor(r^{-1}(x)) | ncolor\}
\end{array}
$$

In other words, “solution” constitutes the set of all combinations of labeled nodes that represent a correct 3-coloration ring. The definition of “solution” is given by (axm2).

### 5.2.3 The “m2-LC1” machine instantiation

In this level, we instantiate the “m2-LC1” as presented in our pattern and we refine the previous machine of the model to obtain a machine that can perfectly specify the rule of the algorithm. First, we add a “t" function which assign a label to each node. Formally this function is specified as follows: $t \in ND \rightarrow COLORS$. Now, with this function we are able to express the relabelling system which assigns a correct 3-coloring to the nodes of a ring. We call “coloring" the new event that shows the high level description of the behavior of each node in the ring. It expresses the

coloring
**any**
$c, ac$
**where**
$grd1 : c \in ND$
$grd2 : t(c) \in t[rg[\{c\}]]$
$grd3 : ac \in COLORS$
$grd4 : ac \notin t[rg[\{c\}]]$
**then**
$act1 : t(c) := ac$
**end**

following : if the center “c" of the star has the same color as one of its two neighbors (we encode the neighbors by “rg[{c}]"), than it changes its label with a new color “ac" which is different from its neighbors color.

## 6 Conclusion

This paper presents a methodological guideline to design *correct-by-construction* distributed algorithms. Our contribution consists in the definition of a common and reusable pattern based on refinement techniques and on the encoding of distributed algorithms by graph rewriting systems. Based on this work, we have developed a new approach called B2Visidia to visualize and to experiment modelling of distributed algorithms. More precisely, we translate the last machine of the model (MLC) into a JAVA implementation for the ViSiDiA tool. Therefore, this work is considered as extension to our work [THMM09] in which we have used the refinement technique of Event-B to propose a unified modular development of distributed algorithms. We have illustrated our method by investigating examples of the distributed computation of spanning trees, and we have implemented them by using the Rodin platform. We plan to implement this pattern as a plug-in in Rodin platform to assist the user in the design of distributed algorithms.

Future directions will investigate:

- the development of other case studies like the naming problem,
- the integration of other proof assistants or proof techniques to improve the automated processing of development
- and the integration of probabilistic reasoning into our approach.

**Acknowledgements:** This work is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

## Bibliography

[Abr10a] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[Abr10b] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Chapter A Mechanical Press Controller. Cambridge University Press, 2010.

[AH07] J.-R. Abrial, S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.* 77(1-2):1–28, 2007.

[AH08] J. R. Abrial, T. S. Hoang. Using Design Patterns in Formal Methods: An Event-B Approach. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*. Pp. 1–2. Springer-Verlag, Berlin, Heidelberg, 2008.

[AR08] P. ANR-RIMEL. Développement d’algorithmes répartis. Technical report, MOSEL, LORIA, Université Henri Poincaré - Nancy 1, ClearSy LABRI, Université de Bordeaux & CNRS, February 2008. http://rimel.loria.fr.

[Bac79] R. Back. On correct refinement of programs. *Journal of Computer and System Sciences* 23(1):49–68, 1979.

[Bac98] R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica* 25:593–624, 1998.

[BM09] N. Benaïssa, D. Méry. Cryptologic protocols analysis using proof-based patterns. In Marchuk (ed.), *Seventh International Andrei Ershov Memorial Conference PERSPECTIVES OF SYSTEM INFORMATICS*. A.P. Ershov Institute of Informatics Systems & Novosibirsk State University, 15-19 June 2009.

[BMMS02] M. Bauderon, Y. Métivier, M. Mosbah, A. Sellami. From local computations to asynchronous message passing systems. Technical report RR-1271-02, LaBRI, 2002.

[CGM08] J. Chalopin, E. Godard, Y. Métivier. Local Terminations and Distributed Computability in Anonymous Networks. In Taubenfeld (ed.), *DISC*. Lecture Notes in Computer Science 5218, pp. 47–62. Springer, 2008.

[CM07a] D. Cansell, D. Méry. EATCS Textbook in Computer Science, chapter The Event-B Modelling Method: Concepts and Case Studies, pp. 33–140. Springer, 2007.

[CM07b] D. Cansell, D. Méry. Incremental Parametric Development of Greedy Algorithms. *Electr. Notes Theor. Comput. Sci.* 185:47–62, 2007.

[CM07c] J. Chalopin, Y. Métivier. An efficient message passing algorithm based on Mazurkiewicz s algorithm. *Fundamenta Informaticae* 80(1):221–246, 2007.

[CM10] J. Chalopin, Y. Métivier. On the power of Synchronisation between Adjacent Processes. *Distributed Computing* 23:177–196, 2010.

[CMZ04] J. Chalopin, Y. Métivier, W. Zielonka. Election, Naming and Cellular Edge Local Computations. In *Graph transformation International Conference on Graph Transformation (ICGT 2004)*. Lecture notes in computer science 3256, pp. 242–256. Springer, Italie, 09 2004.
http://hal.archives-ouvertes.fr/hal-00308119/en/

[GHJ+94] E. Gamma, R. Helm, R. Johnson, R. Vlissides, P. Gamma. *Design Patterns : Elements of Reusable Object-Oriented Software design Patterns*. Addison-Wesley Professional Computing, 1994.

[HFA09] T. S. Hoang, A. Furst, J.-R. Abrial. Event-B Patterns and Their Tool Support. *Software Engineering and Formal Methods, International Conference on* 0:210–219, 2009.
doi:http://doi.ieeecomputersociety.org/10.1109/SEFM.2009.17

[LB08] M. Leuschel, M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer*, pp. –, 2008.

[LMS99] I. Litovsky, Y. Métivier, E. Sopena. Graph relabelling systems and distributed algorithms. In Ehrig et al. (eds.), *Handbook of graph grammars and computing by graph transformation*. Volume 3, pp. 1–56. World Scientific, 1999.

[Maz97] A. Mazurkiewicz. Distributed Enumeration. In *Inf. Processing Letters*. Pp. 23–3. 1997.

[MO04] M. Mosbah, R. Ossamy. A Programming Language for Local Computations in Graphs: Computational Completeness. In IEEE (ed.), *Proceedings of the 5th. Mexican International Conference in Computer Science Colima Mexico 20-24 September*. Pp. 12–19. Computer Society, 2004.
http://www.labri.fr/publications/l3a/2004/MO04

[Mos09] M. Mosbah. ViSiDiA. http://visidia.labri.fr, 2009.

[Mér09] D. Méry. Refinement-Based Guidelines for Algorithmic Systems. *International Journal of Software and Informatics* 3(2-3):197–239, 2009-09.

[Pro04] Project RODIN. Rigorous Open Development Environment for Complex Systems. http://rodin-b-sharp.sourceforge.net/, 2004. 2004–2007.

[Reh09] J. Rehm. Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method. *International Journal on Software Tools for Technology Transfer (STTT)*, 2009.
http://hal.inria.fr/inria-00336624/en/

[THMM09] M. Tounsi, A. Hadj Kacem, M. Mosbah, D. Méry. A Refinement Approach for Proving Distributed Algorithms : Examples of Spanning Tree Problems. In *Integration of Model-based Formal Methods and Tools - IM_FMT'2009 - in IFM'2009*. Düsseldorf Allemagne, 02 2009.
http://hal.archives-ouvertes.fr/hal-00361933/en/