A Programming Language For Local Computations in Graphs. (L.I.D.A.E.Lo.C.[1])

Mohamed Mosbah, Rodrigue Ossamy
LaBRI - University of Bordeaux I
351 Cours de la Libération
33405 - Talence
{mosbah,ossamy}@labri.fr

March 2, 2004

[1]Language for Implementing Distributed Algorithms Encoded by Local Computations. Instead of using the whole acronym, we will simply refer to our language with the *LIDiA* .

Contents

Chapter 1

abstract

This paper presents a new programming language for implementing distributed algorithms encoded by means of local computations [32]. The language, called *LIDiA* , is based on a *two-level* transition system model: the first level is used to specify the behavior of each single component, whereas the second level captures their interactions. Transitions are basically expressed in a *precondition-effect* style. Furthermore, *LIDiA* depends on a logic $\mathcal{L}^*_\infty$ that is used to express the preconditions of each transition. The logic $\mathcal{L}^*_\infty$ is an extension of first-order logic by means of new counting quantifiers and additional computation symbols. We illustrate the different aspects of implementations in *LIDiA* using various classical distributed algorithms.

Chapter 2

Introduction

2.1 The Model

Local computations on graphs, and particularly graph relabelling systems, have been introduced in [4] as a suitable tool for encoding distributed algorithms, for proving their correctness and for understanding their power. In this model, a network is represented by a graph which vertices denote processors, and edges denote communication links. The local state of a processor (resp. link) is encoded by the label attached to the corresponding vertex (resp. edge). A relabelling rule is a rewriting rule which has the same underlying fixed graph for its left-hand side and its right-hand side, but with an update of the labels. 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 change according to some specific *computation rules*. Using this solid theoretical basis, we have developed the *LIDiA* programming language for expressing and programming distributed algorithms [32].

2.2 Related Models

The language *LIDiA* can be viewed as a coordination language that allows users to design and implement concurrent systems. The model provides a precise way of describing and reasoning about system components that interact with each other and that operate at different speeds. Our approach consists of defining an operational model for *LIDiA* , based on a two-level transition system. The first level consists of a number of transition systems, each of which defines the behaviour of a single process. The behaviour of a process depends on the states of its neighbourhood. The second level consists of a single transition system that defines the interactions among the first-level transition systems. Multi-level transition systems were first used

to define the formal semantic of coordination languages in [40] and [41]. This formalism is not specific to any language nor to control-oriented coordination. Indeed, multi-level transition systems are much more general and seem to be suitable for formalizing data-oriented coordination models and languages as well, as illustrated in [8]. Moreover, *LIDiA* is not based on any automaton model. Our goal was simply to develop a programming language which should be as simple as possible and could express all transition definitions needed to describe any given distributed algorithm encoded by means of local computations.

Although, the multi-level transition system is powerful enough to model distributed algorithms, the power of *LIDiA* is addicted to the descriptive complexity of the logic $\mathcal{L}^*_\infty$. Similar logics have been studied by other authors, and shown to be particularly robust by [38, 17, 22]. The most important aspect of the language $\mathcal{L}^*_\infty$ is, among other things, its ability to express counting. In fact, counting is a fundamental operation of numerous algorithms. Counters constitute also an essential primitive of query languages. In relational databases, practical query languages, such as SQL, provide counters as built-in functions of the languages. Counters map relations to integers. They are of great importance from a practical point of view. Moreover, counters raise challenging theoretical problems. Logical languages generally lack the ability to express counting, though it is very easy to count on any computational device [1].

2.3 Other Models

The *LIDiA* model is similar to the labelled transition system models used to define semantics for process algebraic languages like CSP [23] and CCS [31]. In contrast to such languages, *LIDiA* does not define parallel composition and each component has only two external actions: *Send* and *Receive*. *LIDiA* is exactly devoted to the design and implementation of distributed algorithms encoded by means of local computations. Other languages for describing concurrent systems are based on several types of automata, with different notions of composition and external behaviour; for instance, TLA [24] and UNITY [6] are based on state automata that combine via shared variables. The communication in *LIDiA* is due to a messages passing system that is encoded in the second level transition system.

Although IOA [15] and *LIDiA* use *transition definition* (guarded commands) consisting of *preconditions* and *effects*, the preconditions in *LIDiA* are exclusively described in the logic $\mathcal{L}^*_\infty$. We have shown that this logic has enough descriptive power to fully described all *PTIME* queries in the structures used in *LIDiA* . Finally, this particularity of $\mathcal{L}^*_\infty$ has help us to state the completeness of *LIDiA* .

Chapter 3

Local Computations for Encoding Distributed Algorithms

In this Section, we give a few definitions of local computations, and particularly of graph relabelling systems. As usual, such a network is represent ed 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 [2, 3]. In order to make the definitions easy to read, we give in the following an example of a graph relabelling system for computing a spanning tree, and an example of local computations for detecting

stable properties. 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):

Whenever an A-labelled node is linked by a 0-labelled edge to an N-labelled node, then the corresponding subgraph may rewrite itself according to the rule.

A sample computation using this rule is given in Figure 3.1. Relabelling steps *may* occur concurrently on disjoint parts on the graph. When the graph is irreducible, i.e no rule can be applied, a spanning tree, consisting of edges labelled 1, is computed.

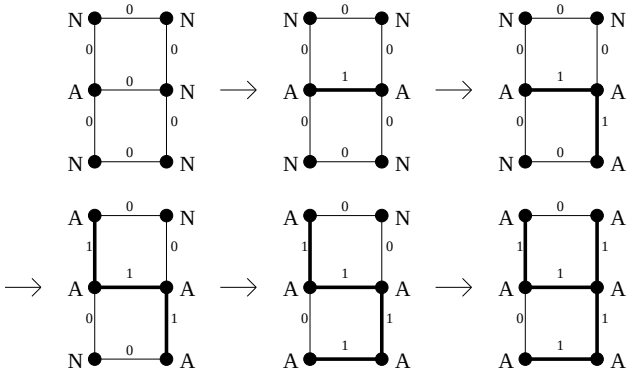


Figure 3.1: Distributed computation of a spanning tree

3.2 Detection of stable properties

The algorithm of Szymanski, Shi and Prywes (SSP's algorithm for short) [44] is a good example to illustrate the notion of local computations.

Consider a distributed algorithm which terminates when all processes reach their local termination conditions, each process is able to determine only its own termination condition. SSP's algorithm detects an instant in which the entire computation is achieved.

Let G be a graph, to each node v is associated a predicate $P(v)$ and an integer $a(v)$. Initially $P(v)$ is false and $a(v)$ is equal to -1. Transformations of the value of $a(v)$ are defined by the following rules.

Each local computation acts on the integer $a(v_0)$ associated to the vertex v_0; the new value of $a(v_0)$ depends on values associated to its neighbours. More precisely, let v_0 be a vertex and let $\{v_1, ..., v_d\}$ the set of vertices adjacent to v_0.
We consider in this Section the following assumption. For each node v, the value $P(v)$ eventually becomes true and remains true for ever.

- If $P(v_0) = false$ then $a(v_0) = -1$;
- if $P(v_0) = true$ then $a(v_0) = 1 + Min\{a(v_k) \mid 0 \leq k \leq d\}$.

This algorithm is useful to detect locally the global termination of a distributed algorithm [30].
A large family of distributed algorithms can be described as local computations, including election, termination detection, computation of a spanning tree [3]. Let us give now a formal definition of local computations.

3.3 Formal definition of local computations

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. 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. 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, λ), where λ: $V(G) \cup E(G) \to L$ is the function labelling vertices and edges. The graph G is called the underlying graph, and the mapping λ is a labelling of G. Let $\mathcal{G}_L$ be the class of graphs labelled over some fixed alphabet L.

Definition 3.3.1 *A graph rewriting relation is a binary relation* $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ *closed under isomorphism. The transitive closure of* $\mathcal{R}$ *is denoted* $\mathcal{R}^*$.
An $\mathcal{R}-$*rewriting chain is a sequence* $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_n$ *such that for every* i, $1 \leq i < n$, $\mathbf{G}_i \ \mathcal{R} \ \mathbf{G}_{i+1}$. *A sequence of length* 1 *is called an* $\mathcal{R}-$*rewriting step (a step for short).*
By "closed under isomorphism" we mean that if $\mathbf{G}_1 \simeq \mathbf{G}$ and $\mathbf{G} \ \mathcal{R} \ \mathbf{G}'$, then there exists a labelled graph $\mathbf{G}'_1$ such that $\mathbf{G}_1 \ \mathcal{R} \ \mathbf{G}'_1$ and $\mathbf{G}'_1 \simeq \mathbf{G}'$.
Definition 3.3.2 *Let* $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ *be a graph rewriting relation.*

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

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

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

2. *A relabelling relation* $\mathcal{R}$ *is local if whenever* $(G, \lambda) \ \mathcal{R} \ (G, \lambda')$, *the labelling* λ *and* λ' *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* $\mathcal{R}-$*normal form of* $\mathbf{G} \in \mathcal{G}_L$ *is a labelled graph* $\mathbf{G}'$ *such that* $\mathbf{G} \ \mathcal{R}^* \ \mathbf{G}'$, *and* $\mathbf{G}' \ \mathcal{R} \ \mathbf{G}''$ *holds for no* $\mathbf{G}''$ *in* $\mathcal{G}_L$. *We say that* $\mathcal{R}$ *is Noetherian if for every graph* $\mathbf{G}$ *in* $\mathcal{G}_L$ *there exists no infinite* $\mathcal{R}-$*relabelling chain starting from* $\mathbf{G}$. *Thus, if a relabelling relation* $\mathcal{R}$ *is Noetherian, then every labelled graph has an* $\mathcal{R}-$*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 3.3.3 *Let* $\mathcal{R}$ *be a relabelling relation. Then* $\mathcal{R}$ *is locally generated if the following is satisfied: For any labelled graphs* (G, λ), (G, λ'), (H, η), (H, η') *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* φ: $V(B_G(v,1)) \longrightarrow V(B_H(w,1))$ *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 $(G, \lambda) \ \mathcal{R} \ (G, \lambda')$ *if and only if* $(H, \eta) \ \mathcal{R} \ (H, \eta')$.

Finally, local computations are the computations defined by a relation locally generated. The reader can find in [3] 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 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 [3]).

Chapter 4

The language *LIDiA*

4.1 The model

Our approach consists of defining an operational model for *LIDiA* that is based on a two-level transition system. The first level consists of a number of transition systems, each of which defines the behavior of a single process. The second level consists of a single transition system that defines the interactions among the first-level transition systems.

We use a set of first-level transition systems to specify processes as autonomous entities that can compute and / or interact with their environment. Thus every step of the computation in such a process may depend not only on the internal state of the process, but also on some input it may obtain from its environment. Such processes are open systems in a sense analogous to Wagner's notion of Interaction Machines [42]. Typically, each such transition system is unbounded and nondeterministic, reflecting the fact that the process it represents is an interactive system; i.e., its unpredictable behavior depends on the input it obtains from an external environment that it does not control. The environment of each process is represented by the set of processes that belong to its neighborhood and by system external actions that could force the execution of a given action in the network. Transition systems, which are structures commonly used in operational semantics, have been used in an uniform and universal way. Every process that exists in *LIDiA* , is modeled as a transition system (in the first-level). Each such transition system describes the potential steps that its corresponding process can take, assuming that it is embedded in an environment that is optimally cooperative.

The details of the internal activity of each process (e.g., its computations) are described by its respective first-level transition system. Most such detail is irrelevant for, and hence unobservable by, the second-level transition system.

The second-level transition system, thus abstract away the semantics of the first-level processes, and is concerned only with their (mutually en-

gaging) externally observable behavior. The external activities of an entire *LIDiA* application are modeled by the second-level transition system. Here, a configuration corresponds to a set of processes each of which is associated with a list of pending messages that have already been broadcast but not yet received. Each second-level transition is defined in terms of transitions reflecting the actions of interacting processes. The second-level transitions are based only on partial view of the whole system, reflecting the true time and space decoupling of processes in a *LIDiA* application.

Further on, in a computational point of view, the second-level transitions are the same for all processors involved in the computation. They represent a formal way how communication links between two processors are implemented.

The model we use in our language is similar to the labeled transition system models used to define semantics for process algebraic languages like CSP [23] and CCS [31]. In particular, those models also define parallel composition in terms of identifying external actions. Other languages for describing concurrent systems are based on several types of automata, with different notions of composition and external behavior; for instance, TLA [24] and UNITY [6] are based on automata that communicate via shared variables. Among all similar models we considered, the notations and model introduced in IOA [15] were very helpful for our work. But instead of representing each process as an *I/O automata*, we simply consider a process as an entity that belongs to a compact system and can perform several rules (first-level system). This departure from the automata model is motivated by the fact that our model has to be as simple as possible. At our actual development step, we do not want users to deal with automata operations nor to perform some invariants proofs with a given theorem prover. We are only interested in computing and visualizing distributed algorithms encoded by means of local computations.

4.2 An informal overview

The *LIDiA* language is designed to allow precise and direct description of distributed algorithms encoded by means of local computations. Since the model we used is a reactive system model rather than a sequential program model, the language reflects this fundamental distinction. That is, it is not a standard sequential programming language with some construct for concurrency and interaction added on; rather, concurrency and interaction are at its core. Two major concepts in *LIDiA* are separation of concerns and anonymous communication. Separation of concerns means that computation concerns are isolated from the communication and cooperative concerns. Anonymous communication means that the processes engaged in communication with each other need not know each other. Furthermore all communication

is asynchronous. In *LIDiA* communications is either through broadcast of events or through point-to-point channel connections which are established between two communicating processes.

The starting point for *LIDiA* was the pseudocode used in earlier works on *Graph Relabelling Rules* and on *I/O automata*. This pseudocode contains, in the case of *I/O automata*, explicit representations of state transition definition in form of *(actions, states, transitions,...)*. Transitions are described using *transition definitions* (TDs) containing *preconditions* and *effects*. This pseudocode has evolved in two different forms: *a declarative style* (see, e.g., [37]), in which effects are described by predicates relating pre- and post-states, and an *imperative style*(e.g., [26]), in which effects are described by simple imperative programs.

Because of our intention to build a formally defined programming language, we have to make some design decisions in order to perform a suitable relationship between our model and the corresponding programming language.

- We use graph data type to symbolize a distributed systems. Each node represents a process and each edge can be seen as a communication link between two processes.

- Each node (respectively edge) has a label that describes its state at a given time.

- Every computing entity in *LIDiA* is a process: vertices and communication links. We will use the word *process* to define a process that represents a network vertex. For processes that represent communication link, we will use the word *edge process* to designate them.

- For our purpose, we allow only the imperative style in each TD(**rule**). Thus, a rule effect may be described entirely by a program. Such imperative descriptions of effects are kept simple, consisting of(possibly nondeterministic) assignments, conditionals, and simple bounded loops. This simplicity makes sense, because transitions are supposed to be executed atomically.

- Variables can be initialized using ordinary assignments and nondeterministic choice statements. The entire initial state may be constrained by predicate.

- The *LIDiA* language can take advantage of some local computations protocols previously introduced and used by Bauderon, Métivier et. al. [3]. These are randomized algorithms which are used to implement local computations in an asynchronous system with asynchronous message passing.

- Each TD corresponds to a **rule** that is represented in a *precondition-effect* style. A **rule** can have additional *choose* parameters, which are not formally part of the action name, but which allow values to be chosen to satisfy the precondition and then used in describing the effect.

- We uses two different sorts of **rules**: *active rule* and *passive rule*. A process uses an *active* rule, when it can decide to execute an action. It uses a *passive* rule, when one of its neighbors processes tells it to execute a given action. Passive and active rules are of prime importance, when dealing with synchronization protocols.

- An important aspect of nondeterministic programming is allowing maximum freedom in the order of action execution. Control over action order is sometimes needed, particularly at lower levels of abstraction where performance requirements may force particular scheduling decisions. For this reason, we have integrated an explicit support for specifying action order in *LIDiA* . Thus, each rule is enhanced with a list of all rules that have a higher order of priorities. If there exists no priority decision between two actions, a random choice is made to designate which rule should be executed.

Languages such as IOA [15], UNITY, MANIFOLD [14], SPECTRUM and TLA are similar to *LIDiA* in that their basic program units are transition definitions with preconditions and effects. However, effects in TLA are described declaratively, effects in UNITY and SPECTRUM are described imperatively and effects in IOA are declared declaratively and imperatively.

4.3 Data types in *LIDiA*

This Section presents an overview of data types that can be used in our language. The list given below is not exhaustive. We take advantage of the data type semantic defined in IOA. A general description of all data types with their respective operations will be described in future papers on *LIDiA* . *LIDiA* enables users to define new data types to define the actions and states of each process. The data types *graph*, *node*, *edge*, *Bool*, *Int*, *Nat*, *Real*, *Char*, and String can appear in *LIDiA* descriptions without explicit declarations. The *graph* data type we used represents parameterized graphs. Thus, any instance of graph contains labels attached to its nodes and edges. Compound data types can be constructed using the following type constructors and used without explicit declarations:

- *Array[I, E]* is an element of elements of type E indexed by elements of type I.

- *Seq[E]* is a finite sequence of elements of type E.

- *Set[E]* is a finite set of elements of type E.
- *Mset[E]* is a finite multiset of element of type E.

Users can define additional data types, as well as redefine built-in types. First, they can explicitly declare enumeration, tuple, and union types analogous to those found in many common programming languages. For example,

- **type** $Color$ = enumeration of red $white$, $blue$
- **type** Msg = tuple of $source$, $dest$: $Process$, $contents$: $String$

For further research, we intend to give the user the possibility to give a specification for the syntax and the semantic of new data types. This specification should be given in a language like the *Larch Shared Language* [19] that is used in [15].

Chapter 5

Description of a Process

Reactive processes are described by specifying their neighborhood, state variables and rules. All of these elements must be present in every process description. Note that the state variables of a process define the label of the corresponding vertex.

5.1 Neighborhood

The neighborhood of a process can be define using the keyword **universe**. It is usually a set of nodes that are stored in a variable B by the expression $B := G.neighborhood()$. With G representing the whole graph network.

5.2 State variables

The declaration of state variables is done using the keyword **Declaration**. We have two different sorts of state variables. The first sort is introduced by the type name **eLabel** and the second one by **nLabel**. These type names have the following meanings:

- The *eLabel* type represents the type of the state variables that characterized the state of each edge (communication link) in the underlying network.
- The type *nlabel* represents the type of the state variables that characterized the state of each node (processor) in the underlying network.

State variables can be initialized using the assignment operator := followed by an expression. State variables are initialized simultaneously and the initialization given for one state variable can not refer to the value of any other state variable. We can also use the keyword **choose** to do a nondeterministic choice to initialize state variables. Once one has initialized the state variables, the underlying network is then enhanced with these type of labels.

This is done by using statement like *G.init*(*nodeLab*, *edgeLab*) where *nodeLab* and *edgeLab* respectively represent instances of the *nLabel* and *eLabel* data types.

5.3 Rules

The liste of rules is introduced by the keywords **active rules** or **passive rules**. Each rule has three parts: **Precondition**, **Relabeling** and **Priorities**.

5.3.1 Precondition

All variables in the precondition must be state variables, be **choose** parameters, or be quantified explicitly in the precondition. If no precondition is given, it is assumed to be true. An action is said to be *enabled* in a state if the precondition for its execution is true in that state. The precondition is expressed in the logical language $\mathcal{L}^*_\infty$ [32]. This language represents an extension of first-order logic by means of counting quantifiers. It has nice properties that helped us to understand the completeness of *LIDiA* .

5.3.2 Relabeling

The relabeling of a rule corresponds to the effect of using the corresponding transition. The Relabeling is generally defined in terms of a (possibly nondeterministic) program that assigns new values to state variables. If the relabeling part is missing, then the corresponding transition has no effect; i.e., it leaves the state unchanged. Passive rules do not have a *Relabeling* part. Rather, they dispose of an *Action* part that should express the fact that any vertex that executes a passive rule is not able to take any relabeling decision.

5.3.3 Priorities

The Priorities part contains a list of rules that have a higher priorities. Let R_i be a given rule that belongs to the priorities-list of rule R. If the preconditions rules R and R_i are satisfied at the same time, then the actions of rule R_i are executed. Thus, the actions of rule R are temporarily "frozen".

5.4 Conditional actions

Sometimes it is necessary for a process to change the labels of some of its neighbours. To deal with such actions, *LIDiA* provides *conditional actions* introduced by the operators **IMPLIES** and **CHOOSE**. The syntax of these operators is described in the following way:
ϕ, **IMPLIES** *Action*

The Action-part of the *IMPLIES* statement is only executed if the logical condition ϕ is true.
Example:
(1). $\forall w \in B(w \neq v_0,$ IMPLIES $L_v[w] := ReceiveFrom(w))$;

In this example, the action *ReceiveFrom(w)* will be executed for all w that satisfy the statement $w \in B \wedge w \neq v_0$. A second example could be expressed by the expression
(2). $\forall w \in B(w \neq v_0),$ IMPLIES $L_v[w] := ReceiveFrom(w))$;
In this case, the *ReceiveFrom(w)* action will only be executed for the last process w that satisfy the condition $w \in B \wedge w \neq v_0$. If we replace the quantifier $\forall$ by the existence quantifier $\exists$, the action will only be executed for the first w that satisfies the given formula ϕ. Note that ϕ is given in the language $\mathcal{L}^*_\infty$.

$x :=$ **CHOOSE** p **IN** ϕ_p
This statement is a random construction that chooses randomly an element p among all elements that satisfy a given formula ϕ_p. This choice is made in an equiprobable way. In this construction, the choosen element is stored in the variable x.

Chapter 6

Communication between processes

Each communication link is represented as an edge process that is in charge of transferring informations between two processes. We use the following three primitives to express the communication between two processes:

- $SendTo(p, m)$: The main goal of this primitive is to send the message m to the neighbor p of a given process i. After a process i has execute this operation, the edge process transmits the message m to the neighbor p.
- $SendToAll(m)$: This primitive executes the actions $SendTo(p, m)$ for all neighbors p of a given process i.
- $ReceiveFrom(p)$: This primitive allows a given process i to receive a message from one of its neighbors p. In *LIDiA* we only deal with reliable communication channels that neither loss nor reorder messages in transit. Furthermore, we give the $ReceiveFrom$ primitive a higher priority than all others. This means that as long as a process is waiting from a message, he can not perform any other actions.

6.1 Communication channel

Each edge process represents a communication channel that can be viewed as a process having exactly two neighbors and which state variable is a four tuple representing the status of four message buffers b_0, b_1, s_0, s_1. An edge process has also two main rules called $Send_0$ and $Send_1$. We now consider a given edge process E_p that should enable the communication between the processes i and j (see Figure 6.1). Each buffer of E_p is a sequence of messages initialized to the empty sequence. When process i execute the primitive $SendTo(j, m)$, the message m is append to the buffer b_0. Once the buffer b_0

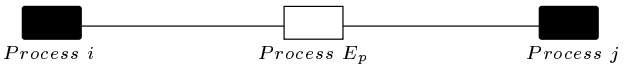


Figure 6.1: An edge process E_p

is not empty, the action (rule) $Send_0$ is executed. This action has the effect of appending the message m, on the head of b_0, to the buffer s_0. Simultaneously, m is deleted from b_0. Note that each edge process E_p always attempts to read and transmit messages that arrive in b_0 and s_0. The buffers b_1 and s_1 are used in the same way for the primitive $SendTo(i, m)$ initiated by process j.

As soon as the buffer s_0 is not empty, the execution of the primitive $ReceiveFrom(i)$ by process j will read the head element of s_0 and consider it as a message sent by process i and actualize the buffer s_0. The same access strategy with buffer s_1 is used for the primitive $ReceiveFrom(j)$ executed by process i.

6.2 Implementation of an edge Process

The implementation of an edge process is similar to the description given in the previous section. For a formal description we give the *LIDiA* program described in figure 6.2. For a better understanding, the set of edge processes

```
Declaration
type Label = tuple of b0, b1, s0, s1: seq < M >;

nodeLab : Label;
Initialization

channel(v0 : node)
univers
B : node_set;
B := G.voisinage(v0);
active rules: Send0, Send1;
Send0:={
precondition
b0.empty() ≠ false;
Relabelling
s0.append(b0.head());
b0.Pop();
Priorities
{};
};
Send1:={
precondition
b1.empty() ≠ false;
Relabelling
s1.append(b1.head());
b1.Pop();
Priorities
{};
};
channel.run(G);
```

Figure 6.2: Implementation of a communication channel process.

has to be seen as an internal representation of communication links. The type M of messages that should be transmitted from a process to an other is set while defining the first-level transition system. But in a general way, the transmitted messages are represented as instances of the type *string*. Further on, the language *LIDiA* supports the use of labels on edges. Each edge process is therefore enhanced with two new buffers and a state variable of type *elabel*. These new variables have the aim to manage and actualize the state changes of the corresponding edge. Figure 6.3 gives a complete description of a communication process in *LIDiA* .

```
Declaration
type Label = tuple of Lab: eLabel,b0, b1,s0,s1,L0,L1: seq < M >;
nodeLab : Label;
Initialization


channel(v0 : node)
univers
B : node_set;
B := G.voisinage(v0);
active rules: Send0, Send1, Change0, Change1;
Send0:={
precondition
b0.empty() ≠ false;
Relabelling
s0.append(b0.head());
b0.Pop();
Priorities
{Change0, Change1; };
};
Send1:={
precondition
b1.empty() ≠ false;
Relabelling
s1.append(b1.head());
b1.Pop();
Priorities
{Change0, Change1; };
};
Change0:={
precondition
L0.empty() ≠ false;
Relabelling
Lab := L0.head();
L0.Pop();
Priorities
{};
};
Change1:={
precondition
L1.empty() ≠ false;
Relabelling
Lab := L1.head();
L1.Pop();
Priorities
{};
};
channel.run(G);
```

Figure 6.3: A communication channel process.

Chapter 7

Randomized Local Elections

In some *LIDiA* implementations, we have to ensure that, at any step, no two adjacent vertices apply one of the relabeling rule at the same time. We solve this problem by using the three randomized procedures studied in [29].

7.1 Implementation of LC_0.

The implementation of LC_0 is the rendezvous. We consider the following distributed randomized procedure. The implementation is partitioned into rounds; in each round each vertex v selects one of its neighbors $c(v)$ at random. There is a rendezvous between v and $c(v)$ if v is the vertex selected by $c(v)$. In this case, we say that v and $c(v)$ are synchronized. When v and $c(v)$ are synchronized there is an exchange of messages by v and $c(v)$. This exchange allows the two nodes to change their labels. Each message for the synchronization mechanism will be a single bit. An implementation in *LIDiA* is given by Program 7.1.1

Each vertex v repeats forever the following actions :

the vertex v selects one of its neighbors $c(v)$ chosen at random;
the vertex v sends 1 to $c(v)$;
the vertex v sends 0 to its neighbors different from $c(v)$;
the vertex v receives messages from all its neighbors.
(There is a rendezvous between v and $c(v)$ if v receives 1 from $c(v)$;*
*in this case a computation step may be done. *)*

Randomized Rendezvous

Program 7.1.1

Declaration
type $nLabel$ = **tuple of** $choice$: ***node***, $stage$: ***int***;
type $eLabel$ = **tuple of** $state$: ***int***;
G : ***graph < nLabel, eLabel >***;
$nodeLab : nLabel$;
$edgeLab : eLabel$;

Initialisation
$edgeLab.state := 0$;
$nodeLab.stage := 0$;
$G.init(nodeLab, edgeLab)$;

Synchronization

$LC_0(v_0 :$ ***node***$)$
univers
B : ***node_set***;
$B := G.neighborhood(v_0)$;

active rules: R_0, R_1, R_2, R_3;
L_v :***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;
$R_0 := \{$
precondition
$v_0.label.stage = 0$;
Relabelling
v, v_1 : ***node***;
$v := choose\ v_1\ in(v_1 \in B \wedge v_1 \neq v_0)$;
$v_0.label.stage := 1$;
$v_0.label.choice := v$;
$\forall v_1 \in B(v_1 \neq v_0, IMPLIES\ SendTo(v_1, v_0.label))$;
Priorities
$\{\}$;
$\}$;
$R_1 := \{$
precondition
$v_0.label.stage = 1$;
Relabelling
$\forall v_1 \in B(v_1 \neq v_0, IMPLIES\ L_v[v_1] := ReceiveFrom(v_1))$;
Priorities
$\{\}$;
$\}$;
$R_2 := \{$
precondition
$v_0.label.stage = 2 \wedge L_v[v_0.label.choice].choice = v_0$;
Relabelling
$v_0.sync := v_0$;
$v_0.label.stage := 0$;
Priorities
$\{\}$;
$\}$;
$R_3 := \{$
precondition
$v_0.label.stage = 2 \wedge L_v[v_0.label.choice].choice \neq v_0$;
Relabelling
$v_0.sync := nil$;
$v_0.label.stage := 0$;
Priorities
$\{\}$;
$\}$;
$\}$;
$LC_0.run(G)$;

LIDiA implementation of *Rendezvous.*

7.2 Implementation of LC_1

Let LE_1 be the local election for implementing LC_1; it is portioned into rounds, and in each round, every processor v selects an integer $rand(v)$ randomly from the set $\{1, ..., N\}$. The processor v sends to its neighbors the value $rand(v)$. The vertex v is elected in $B(v,1)$ if for each vertex w of $B(v,1)$ different from v : $rand(v) > rand(w)$. In this case a computation step on $B(v,1)$ is allowed : the center is able to collect labels of the leaves and to change its label. Program 7.3.1 presents a possible *LIDiA* implemetation of this algorithm.

Each vertex v repeats forever the following actions :

the vertex v selects an integer $rand(v)$ chosen at random;
the vertex v sends $rand(v)$ to its neighbors;
the vertex v receives integers from all its neighbors.
(The vertex v is elected if $rand(v)$ is strictly greater than integers received by v; in this case a computation step may be done on $B(v,1)$. *)*

Randomized LE_1-Elections.

7.3 Implementation of LC_2

Let LE_2 be the local election for implementing LC_2; as in the LC_1 case, it is portioned into rounds. In each round, every processor v selects an integer $rand(v)$ randomly from the set $\{1, ..., N\}$.

The processor v sends to its neighbors the value $rand(v)$. When it has received from each neighbor an integer, it sends to each neighbor w the max of the set of integers it has received from neighbors different from w. The vertex v is elected in $B(v,2)$ if $rand(v)$ is strictly greater than $rand(w)$ for any vertex w of the ball centered on v of radius 2; In this case a computation step may be done on $B(v,1)$. During this computation step there is a total exchange of labels by nodes of S_v, this exchange allows nodes of S_v to change their labels. One *LIDiA* implementation of this algorithm is given in Program 7.3.2.

Each vertex v repeats forever the following actions :

the vertex v selects an integer $rand(v)$ chosen at random;
the vertex v sends $rand(v)$ to its neighbors;
the vertex v receives messages from all its neighbors;
let Int_w the max of the set of integers that v has received from vertices different from w;
the vertex v sends to each neighbor w Int_w;
the vertex v receives integers from all its neighbors;
(There a LE_2-Election in $B(v,2)$ if $rand(v)$ is strictly greater than all integers received by v; in this case a computation step may be done*

on $B(v,1)$. **)*

Randomized LE_2-Election.

Analysis of these algorithms has been done in [28, 29]. It is based on the consideration of *rounds*: in order to measure the performance of the algorithm in terms of the number of rendezvous or local elections taking place, it is assumed that at some instant each node sends and receives messages. Thus this parameter of interest, which is the (random) number of rendezvous or local elections, is the maximal number (i.e. under the assumption that all nodes are active) authorized by the algorithm. It has been proved that these three algorithms are Las Vegas algorithms.

Program 7.3.1

Declaration
type $nLabel$ = **tuple of** $choice$: ***node***, $stage$: ***int***;
type $eLabel$ = **tuple of** $state$: ***int***;
G : ***graph < nLabel, eLabel >***;
$nodeLab$: $nLabel$;
$edgeLab$: $eLabel$;

Initialization
$edgeLab.state := 0$;
$nodeLab.stage := 0$;
$G.init(nodeLab, edgeLab)$;

Synchronization

$LC_1(v_0$: ***node***$)$
univers
B : ***node_set***;
$B := G.neighborhood(v_0)$;
active rules: R_0, R_1, R_2, R_3;
L_v :***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;
$R_0 := \{$
precondition
$v_0.label.stage = 0$;
Relabelling
n, n_1 : ***int***;
v : ***node***;
$n := choose\ n_1\ in(n_1 \in \mathbb{N})$;
$v_0.label.choice := n$;
$v_0.label.stage := 1$;
$SendToAll(v_0.label)$;
Priorities
$\{\}$;
$\}$;
$R_1 := \{$
precondition
$v_0.label.stage = 1$;
Relabelling
v_1 : ***node***;
$\forall v_1 \in B(v_1 \neq v_0,\ IMPLIES\ L_v[v_1] := receiveFrom(v_1)$;
$v_0.label.stage := 2$;
Priorities
$\{\}$;
$\}$;
$R_2 := \{$
precondition
$v_0.label.stage = 2 \wedge v_0.label.choice > max\ i\ in(\exists v_1 \in B(i = L_v[v_1].choice))$;
Relabelling
$v_0.sync := v_0$;
$v_0.label.stage := 0$;
Priorities
$\{\}$;
$\}$;
$R_3 := \{$
precondition
$v_0.label.stage = 2 \wedge (\exists v_1 \in B(v_0.label.choice < L_v[v_1].choice))$;
Relabelling
$v_0.sync := nil$;
$v_0.label.stage := 0$;
Priorities
$\{\}$;
$\}$;
$\}$;
$LC_1.run(G)$;

LIDiA implementation of LC_1.

Program 7.3.2

Declaration
type $nLabel$ = **tuple of** $choice$: ***node***, $stage$: ***int***;
type $eLabel$ = **tuple of** $state$: ***int***;
G : ***graph < nLabel, eLabel >***;
$nodeLab$: $nLabel$;
$edgeLab$: $eLabel$;
Initialization
$edgeLab.state := 0$;
$nodeLab.stage := 0$;
$G.init(nodeLab, edgeLab)$;

$LC_2(v_0$: ***node***$)$
univers
B : ***node_set***;
$B := G.neighborhood(v_0)$;
active rules: R_0, R_1, R_2, R_3, R_4;
L_v, M_v :***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;
$M_v.init(B, nodeLab)$;
$R_0 := \{$
precondition
$v_0.label.stage = 0$;
Relabelling
n, n_1 : ***int***;
$n := choose\ i\ in(i \in \mathbb{N})$;
$v_0.label.choice := n$;
$v_0.label.stage := 1$;
$SendToAll(v_0.label)$;
Priorities
$\{\}$;
$\}$;
$R_1 := \{$
precondition
$v_0.label.stage = 1$;
Relabelling
$\forall v_1 \in B(v_1 \neq v_0,\ IMPLIES\ L_v[v_1] := receiveFrom(v_1)$;
$v_0.label.stage := 2$;
Priorities
$\{\}$;
$\}$;
$R_2 := \{$
precondition
$v_0.label.stage = 2$;
Relabelling
$\forall v \in B(v \neq v_0,\ IMPLIES\ L_v[v].choice := max\ i\ in(i \in \mathbb{N} \wedge \exists v_1 \in B(v_1 \neq v_0 \wedge v_1 \neq v \wedge i = L_v[v_1].choice)))$;
$\forall v_1 \in B(v_1 \neq v_0,\ IMPLIES\ SendTo(v_1, L_v[v_1])$;
$v_0.label.stage := 3$;
Priorities
$\{\}$;
$\}$;
$R_3 := \{$
precondition
$v_0.label.stage = 3$;
Relabelling
$\forall v_1 \in B(v_1 \neq v_0,\ IMPLIES\ L_v[v_1] := receiveFrom(v_1)$;
$v_0.label.stage := 4$;
Priorities
$\{\}$;
$\}$;
$R_4 := \{$
precondition
$v_0.label.stage = 4 \wedge\ \forall v \in B(v_1 \neq v_0, \wedge v_0.label.choice > L_v[v_1].choice)$;
Relabelling
$v_0.sync := v_0$;
$v_0.label.stage := 0$;
Priorities
$\{\}$;
$\}$;
$R_5 := \{$
precondition
$v_0.label.stage = 4 \wedge\ \exists v \in B(v_1 \neq v_0, \wedge v_0.label.choice < L_v[v_1].choice)$;
Relabelling
$v_0.sync := nil$;
$v_0.label.stage := 0$;
Priorities
$\{\}$;
$\}$;
$\}$;
$LC_2.run(G)$;

Chapter 8

Implementation examples

8.1 Example: Spanning tree computation

We illustrate our model, as well as the use of the language *LIDiA* to describe distributed algorithms, by two simples computation examples. First of all we will tackle the computation of a spanning tree in a graph. Program 8.1.1 is a simple *LIDiA* implementation to compute a spanning tree in a graph. The algorithm can be depicted in the following way.

We assume that a unique vertex has initially label A, all other vertices having label N and all edges having label 0. At each step of the computation, an A-labeled vertex u may activate any of its neutral neighbors, say v. In that case, u keeps its label, v becomes A-labeled and the edge $\{u, v\}$ becomes 1-labeled. Thus, several vertices may be active at the same time. Concurrent steps will be allowed provided that two such steps involve distinct vertices. The computation stops as soon as all the vertices have been activated. The spanning tree is then given by the 1-labeled edges. The algorithm may be encoded by the graph relabeling system $\mathcal{R}_1 = (L_1, I_1, P_1)$ defined by $L_1 = \{N, A, 0, 1\}$, $I_1 = \{N, A, 0\}$, and $P_1 = \{R\}$ where R is the following relabeling rule:

Every *LIDiA* program consists of four parts. General and global variables are declared and initialized in the first and second parts. The types *nLabel* and *eLabel* are introduced to define the kind of labels that will be used on vertices and edges. The third part defines the sort of local synchronization protocol that is used in the algorithm. Program 8.1.1 uses the LC_1 local computation protocol to permit trouble-free state transitions. This protocol synchronizes the vertices that belong to the same star. Only the center of the synchronized star can change its label and the labels of the edges that are adjacent to it.

On the other hand, transitions are given in precondition/effect style and are represented by the active relabeling rule R_0 and by the passive rule P_0. All vertices that are not center of a synchronized star execute the passive rule by sending their own label to all their neighbors that are center of any synchronized star. The active rule R_0 is only executed by vertices that are center of a synchronized star. A node v_0 apply R_0 if it is N-labeled and if it has a neighbor that is A-labeled. The consequences of the application of rule R_0 are represented by the relabeling of edge $[v_0, w]$ that becomes 1-labeled and by the fact that v_0 becomes A-labeled. *LIDiA* provides a structure that allows to set an execution priority between two rules. For each rule, this structure is given as a list of rules that has higher execution priorities.

Note that all the rules described in Program 8.1.1 are performed by all the vertices v_0 that belong to the graph G. In *LIDiA* we do not have to specify one list of rules for each process. Rather, we take advantage of the ability of *LIDiA* to define a list of transitions that will be valid for all network's processes. Thus, these rules can be performed simultaneously by different vertices at the same time. It is also possible to define actions that are only executable for some identified processes. In this case we have to define rules that will only be executed by non identified processes. In general our language deals with anonymous networks.

Program 8.1.1 (Spanning Tree computation)

```
Declaration
type nLabel = tuple of var: string;
type eLabel = tuple of var: int;
G : graph < nLabel, eLabel >;
edgeLab : eLabel;
nodeLab : nLabel;

Initialization
edgeLab.var = 0;
nodeLab.var = 'N';
G.init(nodeLab, edgeLab);
v : node;
v := G.choose_node();
v.label.var = 'A';

Synchronization
Synchro_LC1();

SpanningTree(v0 : node) :
universe
B : node_set;
B := G.neighborhood(v0);

active rules: R0;
Lv : node_array < nLabel >;
Lv.init(B, nodeLab);
∀w ∈ B(w ≠ v0, IMPLIES Lv[w] := ReceiveFrom(w));
}

R0 :={
precondition
w: node
v0.label.var = 'N' ∧
∃w ∈ B(w ≠ v0 ∧ Lv[w].var = 'A' ∧ [v0, w].var = 0);
Relabeling
v0.label.var := 'A';
[v0, w].var := 1;
Priorities
{};
};
passive rules:P0;
R0 :={
```

precondition
$v_0.sync \neq v_0$;
Action
$v : node$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,\ IMPLIES\ SendTo(v, v_0.label))$;
Priorities
{};
};
$SpanningTree.run(G)$;

8.2 Example: Drinking philosophers problem

In this section we attend to give an implementation of the *drinking philosophers problem*. This problem was introduced by Chandy and Misra in [5] as a generalization of the *dinning philosophers problem* [9]. This problem can be formulated as followed.

We consider a network of processes sharing a set of resources. It is represented as a connected undirected graph where vertices denote processes and edges represent conflicts between processes. For the *drinking philosophers problem* one has to deal with case where a process needs to have access to all its resources to do any computation. Any algorithm which solves this problem has to ensure the following properties:

- No shared resource can be accessed by two processes at the same time. Thus, the algorithm ensures the *mutual exclusion* on shared resources,
- If two processes p_1 and p_2 do not share a resource, and hence are not adjacent in the underlying graph, then they can access their resources independently, and possibly at the same time. The algorithm ensures therefore the *concurrency* property.
- If a resource is asked by two processes p_1 and p_2 and if p_1 formulates its request before p_2, then p_1 must enter the resource before p_2, this is the so called *ordering* property,
- If a process p asks to have access to all needed resources, at last p must obtain this access. This is the *liveness* property.

The implementation we will give later is based on an algorithm described by M. Mosbah, A. Sellami and A. Zemmari in [34]. Their formulation is based on graph relabeling system and allows an effortless representation in the model of *LIDiA* .

Each process can be in one of the three states: *tranquil*, *thirsty* or *drinking*. These states will be respectively encoded by the labels T, Th and D. One also needs to manage the order of process requests. This is done by using another label which is an integer corresponding to the rank of the request. Therefore, each vertex will be labeled by (X, i) where $X \in \{T, Th, D\}$ and an integer i.

Initially all the vertices of the graph are *tranquil* (this is encoded by the label $(T, -1)$). At each step of the computation, a $(T, -1)$-labeled vertex u may ask to obtain exclusive access to all its critical sections. Thus, it becomes *thirsty*. In this case, u changes its label to $(Th, i + 1)$ where i satisfies the relation $i = max\{k_v | v \in B(u, 1)$ and $\mathcal{L}(v) = (X, k_v), X \in \{T, Th, D\}\}$. So, u has a higher access priority (*order*) than all its neighbors. In a complete graph, this *order* can be seen as an universal time since only one node can change its label to Th at a given time. In this case, the system computes the mutual exclusion.

If a $(Th, i)-$labeled vertex u, has no neighbor in the critical sections (label $(D, -1)$) and no $(Th, j)-$labeled neighbor where $j < i$ (u has a lower priority), then u can enter the critical sections. Thus, u will be $(D, -1)-$labeled.

Once, the vertex in the critical sections has terminated its work, it returns to the *tranquil* state and becomes $(T, -1)-$labeled.

The algorithm may be encoded by the graph relabeling system $\mathcal{R} = (L, I, P)$ defined by $L = \{\{T, Th, D\} \times [-1..\infty]\}$, $I = \{(T, -1)\}$, and $P = \{R_1, R_2, R_3\}$ where R_1, R_2 and R_3 are the relabeling rules given in Figure 8.1:

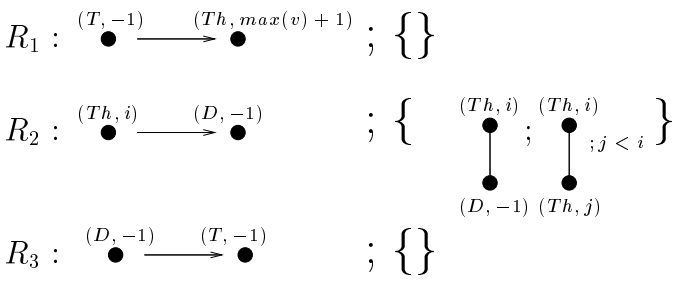


Figure 8.1: Relabeling rules for the Drinking philosophers problem

Program 8.2.1 (Drinking philosophers problem)

```
Declaration
type nLabel = tuple of state: string, rank: int;
type eLabel = tuple of var: int;
G : graph < nLabel, eLabel >;
edgeLab : eLabel;
nodeLab : nLabel;

Initialization
nodeLab.state = 'T';
nodeLab.rank = −1;
G.init(nodeLab, edgeLab);
v : node;
v := G.choose_node();
v.label.var = 'A';

Synchronization
Synchro_LC1();

DrinkingPhilo(v0 : node) :
universe
B : node_set;
```

$B := G.neighborhood(v_0)$;

active rules: R_1, R_2, R_3;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;
$\forall w \in B(w \neq v_0,\ IMPLIES\ L_v[w] := ReceiveFrom(w))$;

$R_1 := \{$
precondition
w: ***node***
$v_0.label.state = 'T' \ \wedge \ v_0.label.rank = -1$;
Relabeling
$v_0.label.state := 'Th'$;
$v_0.label.rank := 1 + Max\ i\ IN(\forall i \in \mathbb{N} \exists u \in B(L_v[u].rank = i))$;
Priorities
$\{\}$;
$\};$
$R_2 := \{$
precondition
w: ***node***
$v_0.label.state = 'Th' \ \wedge \ \forall u \in B \neg(L_v[u].state = 'D' \ \wedge \ L_v[u].rank = -1 \ \vee$
$L_v[u].state = 'Th' \ \wedge \ L_v[u].rank < v_0.label.rank)$;
Relabeling
$v_0.label.state := 'D'$;
$v_0.label.rank := -1$;
Priorities
$\{\}$;
$\};$
$R_3 := \{$
precondition
w: ***node***
$v_0.label.state = 'D' \ \wedge \ v_0.label.rank = -1$;
Relabeling
$v_0.label.state := 'T'$;
$v_0.label.rank := -1$;
Priorities
$\{\}$;
$\};$
passive rules:P_0;
$R_0 := \{$
precondition
$v_0.sync \neq v_0$;
Action
$v : node$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,\ IMPLIES\ SendTo(v, v_0.label))$;
Priorities
$\{\}$;
$\};$
$DrinkingPhilo(G)$;

Chapter 9

Conclusions and Perspectives

Several aspects of the *LIDiA* model make it appropriated to describe and verify distributed algorithms. This language is based on the one hand on set-theoretic mathematics. On the other hand its power and expressiveness relied on the logical language $\mathcal{L}^*_\infty$ that has more expressive power than first-order logic. Furthermore, *LIDiA* is a nondeterministic language in the sens that actions that satisfy a given precondition are executed randomly. This particularity allows distributed systems to be described in their most general forms. In our earlier work [32], we stated the computational completeness of the *LIDiA* language. In fact, every computable distributed problems that could be represented in a precondition-effect style can be computed by a *LIDiA* program.

Future work involves the development of a code generator that could automatically generate real distributed code in a standard programming language like C++, Java, ML or MPI from a *LIDiA* program. Actually, the code generation schemes produces runnable version of *node automata* that can communicate via pre-existing communication services such as TCP or MPI. [35], which are modeled by *channel automata*. This code generator will be embedded later in the *ViSiDiA* platform [33]. Thus, it will help us to verify the generated code. By insisting that *LIDiA* programs from which we generate code match the available computing hardware and communication services, and by requiring the node programs to tolerate input delays, we hope to achieve a faithful implementation without using any non-local synchronization.

Appendix A

The Logic $\mathcal{L}^*_\infty$

In this Section we introduce some extensions of first-order logic that are necessary to understand how the language $\mathcal{L}^*_\infty$ is built. These are fixed-point logics, infinitary logic and infinitary logic with counting. Throughout the rest of this chapter, we will assume that the reader is already familiar with the basic concepts of first-order logic and fixed-point logics as the definition of formulae and how the notion of truth is defined. Our principal reference is "Model Theory" by Chang and Keisler [7].

First of all, recall that infinitary logic $\mathcal{L}_{\infty\omega}$ is the extension of first-order logic where infinite disjunctions and conjunctions of formulas are also allowed. It is well known that any(isomorphism closed) class $\mathcal{C} \subseteq STRUCT[\sigma]$ can be defined in $\mathcal{L}_{\infty\omega}$ (where $STRUCT[\sigma]$ denotes the class of finite σ-structures). Interest of this logic comes from its fragments which have weaker expressive power. One such fragment is $\mathcal{L}^k_{\infty\omega}$ where only k distinct variables, free or bound, are allowed. The finite variable logic $\mathcal{L}^\omega_{\infty\omega}$ is then the union of $\mathcal{L}^k_{\infty\omega}$ over all natural numbers k. Fixed-point logics (least inflationary logic, partial fixed-point logics and transitive closure logic) can all be embedded into $\mathcal{L}^\omega_{\infty\omega}$. It is also easy to see that $\mathcal{L}^\omega_{\infty\omega}$ can not express certain counting properties, such as parity of cardinality. For an extensive study of this logic, see e.g. [13].

For our purpose, we use a logic denoted by $\mathcal{L}^*_\infty$. This logic is obtained by first adding counting terms, counting quantifiers to the logic $\mathcal{L}^\omega_{\infty\omega}$ over two-sorted structures (the second sort being interpreted as $\mathbb{N}$), and then restricting it to formulae of finite rank. The idea of using the set of natural numbers in the two-sorted structure is influenced by meta-finite model theory of [11]. Similar extensions exist in the literature [10, 17, 39], but they restrict the logic by means of the numbers of variables, which still permits fixed-point computation. In contrast, following [20, 22], we restrict the logic by requiring the rank of a formula be finite (where the rank is defined as quantifier rank, except that it does not take into account quantifiers over $\mathbb{N}$), thus putting no limits at all on the available arithmetic.

A.1 The alphabet of $\mathcal{L}^*_\infty$

The conditions and rules of building terms and formulas are the same as in first-order logic. The set of free variables is also defined in the same way as in the first-order logic. Furthermore, the alphabet of $\mathcal{L}^*_\infty$ is obtained by adding the following counting terms and quantifiers to the symbols of first-order logic.

- Counting Quantifiers: $\exists_{\boldsymbol{i}}$, $\exists^{\perp}_{\boldsymbol{i}}$
- Counting Terms: $\exists_{\mathbf{1}}$, $\exists_{\mathbf{2}}$, $\exists_{\mathbf{3}}$,... $\exists^{\perp}_{\mathbf{1}}$, $\exists^{\perp}_{\mathbf{2}}$, $\exists^{\perp}_{\mathbf{3}}$,...
- Equality symbol: $=, \neq$
- Arithmetic functions: $+, -, \times, \div$, $\underline{\mathbf{m}}\boldsymbol{od}$, $\boldsymbol{div}$, $\boldsymbol{exp}$, $\boldsymbol{Fak}$...

All the arithmetic help functions have the intuitive meaning. The counting quantifier $\exists_{\boldsymbol{i}}$ is satisfied if the set of elements, that satisfy a given formula ϕ, has a cardinality greater or equal to i. On the other hand, the formula $\exists^{\perp}_i x\phi(x,i)$ is *true*, if there are exactly i elements x that satisfy $\phi(x)$. We have to notice that the variable i is an element of the set of natural numbers.
Example A.1.1 *With the logic $\mathcal{L}^*_\infty$ we can express the fact that a given set ψ has an even cardinality. This is done by the following expression:*

$$\exists^{\perp}_i x(x \in \psi \ \wedge \ i \bmod 2 = 0);$$

A.2 Semantic of $\mathcal{L}^*_\infty$

Although most of the basic theory was developed for arbitrary structures, the interesting results only speak about finite ones. So we restrict our attention to finite structures with finite vocabularies, unless it is explicitly stated otherwise. Furthermore, we always assume classes of structures to be closed under isomorphism. In the same sense that sentences of a logic define classes of structures, formulas with free variables define queries.
We abbreviate first-order logic by *FOL*. $\mathcal{L}^*_\infty$ is a two-sorted logic, with the second sort being the sort of natural numbers. That is, a structure $\mathbb{A}$ is of the form

$$\begin{aligned}\mathbb{A} = \quad & \langle \{v_1, \cdots, v_m\}, \{1, \cdots, n\}, <, =, \neq, 0, 1, \text{true}, \text{false}, MIN, MAX, \\ & R_1^{\mathbb{A}}, \cdots, R_l^{\mathbb{A}}, f_1^{\mathbb{A}}, \cdots, f_k^{\mathbb{A}} \rangle.\end{aligned}$$

Here the relations $R^{\mathbb{A}}$ are defined on the vertices domain $\{v_1, \cdots, v_m\}$, while on the numerical domain $\{1, \cdots, n\}$ one has constants 0 and 1. The set $\{v_1, \cdots, v_m\}$ even represents the neighborhood of a node v_0. In essence, any vertex u can only see the part of the general domain of $\mathbb{A}$ that is represented by its neighbourhood. furtheron, the added universe of numbers gives us the

ability to do some arithmetic on the side as we express a property of the input structures. In this paper, we will assume that all the structures are equipped with numbers unless we explicitly state otherwise.

A.3 The Satisfaction Relation

The satisfaction relation makes precise the notion of a formula being true under an interpretation. Let D be the domaine of our logic, I be an interpretation relation, $\mathbb{A} = \langle D, I \rangle$ be an interpretation structure, g be an assignment in $\mathbb{A}$ and ϕ be a formula. If ϕ can be represented in the FOL then the satisfaction relation of FOL can be used on ϕ. If ϕ contains new introduced quantifiers, then the satisfaction relation of ϕ by g in $\mathbb{A}$, is given by the following rules:
Counting Quantifier

$$\mathbb{A} \models \exists_i x\phi[g] \quad \Leftrightarrow \quad \mathbb{A} \models \phi[g[x/d]] \text{ for at least } i \text{ elements } d \in D \text{ with } i \leq |D|.$$

$$\mathbb{A} \models \exists_i^{\perp} x\phi[g] \quad \Leftrightarrow \quad \mathbb{A} \models \phi[g[x/d]] \text{ for exactly } i \text{ elements } d \in D \text{ with } i \leq |D|.$$

The satisfaction of counting terms is a special case of the above satisfaction relations.

Appendix B

Computational Completeness

B.1 Descriptive Complexity of $\mathcal{L}^*_\infty$

The main task of this section is to determine the descriptive complexity of the logic $\mathcal{L}^*_\infty$. More precisely, we want to find out the main complexity class that is captured by our logic. We look for results saying that, on a certain domain $\mathcal{D}$ of structures, the logic $\mathcal{L}^*_\infty$ *captures* the complexity class PTIME. This means that the following Definition is satisfied.

Definition B.1.1

(1) *For every fixed sentence $\phi \in \mathcal{L}^*_\infty$, the data complexity of evaluating ϕ on structures from $\mathcal{D}$ is a problem in the complexity class PTIME.*

(2) *Every property of structures in $\mathcal{D}$ that can be decided with complexity PTIME is definable in the logic $\mathcal{L}^*_\infty$.*

As a matter of course, the domain $\mathcal{D}$ and the corresponding structures will be defined as introduced previously.

We assume that the reader is familiar with the basics of complexity theory and has heard of some of the common complexity classes, such as LOGSPACE, PTIME, NPTIME and PSPACE. We assume further that the reader has knowledge about the principles of logics such as *fixed-point logic*, *least fixed-point logic(LFP)* and *inflationary fixed-point logic(IFP)*. Fixed-point logic is an extension of first-order logic designed to reflect the power of induction. There are several formalizations which are not in general equivalent, but the differences are of no concern to us. This is also justified by the results of Gurevich and Shelah [43] stating that many different definitions of fixed-point logic coincide for finite structures. We refer the reader to [13] for a detailed presentation of the basic material of this section.

B.1.1 Logic with counting

From the point of view of expressiveness, *FOL* has two main deficiencies: It lacks the power to express anything that requires recursion(The simplest example is transitive closure) and it can not count, as witnessed by the impossibility to express that a structure has even cardinality. A number of logics add recursion in one way or another to *FOL*, notably the various forms of fixed-point logics. On ordered finite structures, some of these logics can express precisely the queries that are computable in PTIME or PSPACE. However, on arbitrary finite structures they do not, and almost all known examples showing this involve counting. While in presence of an ordering, the ability to count is inherent in fixed-point logic, hardly any of it is retained in its absence.

Therefore Immerman proposed to add counting quantifiers to logics and asked whether a suitable variant of fixed-point logic with counting would suffice to capture PTIME. Meanwhile, fixed-point logic with counting has turned out to be an important and robust logic, that defines natural level of expressiveness and allows to capture PTIME on interesting classes of structures.

There are different ways of adding counting mechanisms to a logic, which are not necessarily equivalent. The most straightforward possibility is the addition of quantifiers of the form $\exists^{\geq 2}$, $\exists^{\geq 3}$, with the obvious meaning. While this is perfectly reasonable for bounded-variable fragments of *FOL* or infinitary logic (see e.g. [16, 39]) it is not general enough for fixed-point logic, because it does not allow for recursion over the counting parameters i in quantifiers $\exists^{\geq i} x$. These counting parameters should therefore be considered as variables that range over natural numbers. This implies indirectly the use of two-sorted structures in most counting logics with the second sort being the set of natural numbers $\mathbb{N}$. We denote by (FOL + C) the *FOL* with counting.

We now define [21] **inflationary fixed-point logic with counting** (IFP + C) and **partial fixed-point logic with counting** (PFP + C) by adding to (FOL + C) the usual rules for building inflationary or partial fixed-points, ranging over both sorts.

Definition B.1.2 *Inflationary fixed-point logic with counting (IFP + C), is the closure of two-sorted FOL under*

(i) *The rule for building counting terms;*

(ii) *The usual rules of FOL for building terms and formulae;*

(iii) *The fixed-point transformation rule: Suppose that $\psi(R, \bar{x}, \bar{\mu})$ is a formula of vocabulary $\tau \cup \{R\}$ where $\bar{x}$, $\bar{\mu}$ and R has mixed arity (k, l), and that $(\bar{u}, \bar{\nu})$ is a $k+l$-tuple of first- and second-sort terms, respectively. Then $[ifp\ R\bar{x}\bar{\mu}.\psi](\bar{u}, \bar{\nu})$ is a formula of vocabulary τ.*

It is clear that counting terms can be computed in polynomial-time. Hence the data complexity remains in PTIME for (IFP + C).

Infinitary logic with counting. Let $C^k_{\infty\omega}$ be the infinitary logic with k variables, $\mathcal{L}^k_{\infty\omega}$, extended by the quantifiers $\exists^{\geq m}$("There exists at least m") for all $m \in \mathbb{N}$. Further, let $C^\omega_{\infty\omega} := \bigcup_k C^k_{\infty\omega}$.

Proposition B.1.1 $(IFP + C) \subseteq C^\omega_{\infty\omega}$.

Due to the two-sorted framework, the proof of this result is a bit more involved than for the corresponding result without counting, but not really difficult. One has first to replace a fixed-point formula by an infinitary formula with counting terms that still lives in the two-sorted framework and then eliminates quantification over number-variables by infinite disjunctions and conjunctions. We refer to [10, 39] for details.

From the definitions of $C^\omega_{\infty\omega}$ and $\mathcal{L}^\omega_{\infty\omega}$ we can deduce that $\mathcal{L}^\omega_{\infty\omega} \subseteq C^\omega_{\infty\omega}$. Furthermore, the next proposition states a relation between the logics $C^\omega_{\infty\omega}$ and $\mathcal{L}^*_\infty$.

Proposition B.1.2 *In finite variable logics, the following relation is valid:*

$$C^\omega_{\infty\omega} \subseteq \mathcal{L}^*_\infty.$$

The proof of this proposition is done by a structural induction. One can also notice that the logic $\mathcal{L}^*_\infty$ can be constructed by augmenting $\mathcal{L}^\omega_{\infty\omega}$ with counting quantifiers, counting Terms, Equality symbol and useful arithmetic functions. This means that in finite structures the above proposition is satisfied.

B.1.2 $\mathcal{L}^*_\infty$ captures PTIME

We present, in this section, the proof of the fact that the logic $\mathcal{L}^*_\infty$ captures PTIME on the class of structure used in *LIDiA* (see section A.2.).

First of all, we can derive corollary B.1.1 from the previous propositions B.1.1 and B.1.2. Thus we have a direct relationship between the language $\mathcal{L}^*_\infty$ and the well-known logic $(IFP + C)$.

Corollary B.1.1 $(IFP + C) \subseteq \mathcal{L}^*_\infty$.

Classical definitions of a logic, such as the notion of *regular logic* (see [12]) do not suffice for our purpose; in addition our logics are supposed to satisfy certain effectivity conditions. Gurevich [18] suggested the following definitions.

Definition B.1.3 *A logic is a pair $(L, \models_L)$ where L is a function that assigns a recursive set $L[\tau]$ of sentences to each vocabulary τ, and $\models_L$ is a binary relation between sentences and structures such that for all $\phi \in L[\tau]$ the class $Mod(\phi) = \{\mathcal{A} \in \mathbb{F} | \mathcal{A} \models_L \phi\}$ is an isomorphism closed class of*

τ-structures. We further say that a class $\mathbb{C}$ of τ-structures is definable in $(L, \models_L)$ if there is an $L[\tau]$-sentence ϕ such that $\mathbb{C} = Mod(\phi)$.

Definition B.1.4 *A clocked PTIME Turing machine is a pair (M, p) where M is a deterministic Turing machine and p a polynomial such that M runs at most $p(n)$ steps on an input of length n.*

Now we can give a better definition that should make clear what we really mean when we say that a logic (effectively) captures a complexity class.

Definition B.1.5 *A logic $\mathbb{L} = (L, \models_L)$ effectively captures a complexity class K on a domain D if the following two conditions hold:*

(i) *Each K-computable class in D is L-definable.*

(ii) *For all vocabularies τ there is a recursive mapping M that associates a clocked K-Turing machine $M(\phi)$ with each sentence $\phi \in L[\tau]$ so that if $Mod(\phi) \in D$ then $M(\phi)$ accepts the language $\mathbb{L}(Mod(\phi))$.*

Clearly, a logic that effectively captures a complexity class captures the class as defined in the previous sections.

Now we are going to state some capturing results concerning the logic (IFP + C).

(i) (Immerman, lander [36], Lindell [25])(IFP + C) effectively captures PTIME on the class of trees.

(ii) (Grohe, Marino [27]) For each $k \geq 1$, (IFP + C) effectively captures PTIME on the class of graphs of tree-width at most k.

Proposition B.1.3 *The language $\mathcal{L}^*_\infty$ effectively captures PTIME on the class of structures defined in section A.2.*

The proof of this proposition is due to the above characteristics of the logic (IFP + C) associated with the consequences of corollary B.1.1. Proposition B.1.3 leads to the main result of this section. In fact, we can use the properties of the logic $\mathcal{L}^*_\infty$ to give a class of distributed problems that can be encoded and solved by *LIDiA* .

Definition B.1.6 *The set Φ defines the class of distributed problems that can be expressed in a precondition/effect style where all the preconditions can be evaluated in PTIME.*

Theorem B.1.1 *For every computable problem $\mathcal{P}$ that belongs to the class Φ there exists a LIDiA program $\mathcal{P}_0$ that computes $\mathcal{P}$.*

We are now ready to state our main result, namely that every computable distributed problem is computed by a *LIDiA* program. We introduce, therefore, some definitions and notations that will help us to bring out the core of the proofs we will state later.

Definition B.1.7 *The class Ω defines the class of all LIDiA programs.*
Definition B.1.8 *The class C defines the class of distributed problems that can be encoded by means of local computations.*
It is clear that each instance of C can be expressed in a *precondition/effect* style and that all nodes and edges of the network have labels that describe their state variables at each computation step.

Theorem B.1.2 *(Completeness of LIDiA) For every computable distributed problem $f \in \mathsf{C}$ there exists a LIDiA program that computes f.*

The proof of theorem B.1.2 depends on the correctness of the following lemmas.

Let $f \in \mathsf{C}$ be a distributed problem and P_f be the set of preconditions that appear in the rules of f. We assume that the network has n vertices.
Lemma B.1.1

$$\forall p \in P_f, p \in PTIME \Rightarrow \exists f_p \in \Omega \text{ that computes } f.$$

Proof B.1.1 *This lemma is a direct consequence of theorem B.1.1. Note that we use the notation "$p \in PTIME$" to express the fact that the data complexity needed for the execution of the query p is polynomial.*
Lemma B.1.2

$$\exists p \in P_f, p \in NPTIME \Rightarrow \exists f_p \in \Omega \text{ that computes } f.$$

Proof B.1.2 *Without loss of generality, let $p \in NPTIME$, $\mathcal{A}_p$ be the class of elements(sets of star graphs of diameter 2) that satisfy p and $\mathcal{O}_r(u, \mathcal{A}_p)$ an oracle that is true if the ball of radius 1 centered on u belongs to the class $\mathcal{A}_p$. Thus, for any node v we can express the precondition p of any rule using the oracle $\mathcal{O}_r(v, \mathcal{A}_p)$. The oracle will return true if the precondition p is satisfied and false otherwise. $\mathcal{O}_r$ can be implemented in LIDiA as an user defined function. Hence, the above Lemma is satisfied. The function $\mathcal{O}_r(u, \mathcal{A}_p)$ is represented outside the precondition of p.*

Theorem B.1.3 *(General Completeness of LIDiA) The class of problems that are computed by LIDiA programs is exactly the class C.*

Because of Theorem B.1.2 and the design of *LIDiA* , Theorem B.1.3 is obvious. In fact, Theorem B.1.2 states the completeness of the language *LIDiA* . This means that any distributed algorithm encoded by local computations can be implemented in *LIDiA* . Furthermore, all computational actions in *LIDiA* are local in the sense that only network computations in a ball of radius 1 are allowed. Thus, any distributed algorithms designed in *LIDiA* can be encoded as a list of local computations rules.

Appendix C

Lexical syntax

We use the following conventions to describe the syntax of *LIDiA* . Uppercase words and symbols enclosed in single quotation marks are terminal symbols in BNF grammar. All other words are nonterminal symbols. If x and y are grammatical units, then the following notations have the indicated meanings.

Notation	Meaning
$x\ y$	an x followed by a y
$x\|y$	an x or a y
$x*$	zero or more x's
$x+$	one or more x's
$x, *$ and $x; *$	zero or more x's separated by commas or semicolons
$x, +$ and $x; +$	one or more x's separated by commas or semicolons

The lexical grammar of *LIDiA* uses the following symbols:

Punctuation marks: , : ; := [ ] { } ( )

Reserved words: Declaration, Initialization, Synchronization, universe, active, rules, precondition, Relabeling, passive, Action, Priorities, choose, const, if, while, do, then, tuple, of, enumeration.

IDENTIFIERS for variables, types, and functions: Sequences of letters, digits, apostrophes, and underscores(exept that two underscores occur consecutively).

OPERATORS: Sequences of the characters $-\ !\ \#\ *\ +\ .\ <\ =\ >\ |$ followed by one of these characters or by an identifier.

The lexical grammar is defined as the four tuple element G_t, where G_t is defined as followed:

$G_t = (N, T, P, \langle Program \rangle)$.
N is the set of nonterminal symbols.
T is the set of terminal symbols, $T \cap N = \emptyset$.
P is a finite set of production rules of the form $\alpha \to \beta$ with $\alpha \in V_+,\ \beta \in V_*,\ V = N \cup T$.
$\langle Program \rangle \in N$ represents the start symbol.

C.1 Declaration Part

$$
\begin{aligned}
\langle Dec \rangle &\to \epsilon | \langle Dec\ suite \rangle ; \\
\langle Dec\ suite \rangle &\to \langle De \rangle | \langle Dec\ suite \rangle ; \langle De \rangle \\
\langle De \rangle &\to \langle const\ De \rangle | \langle var\ De \rangle | \langle type\ De \rangle \\
\langle const\ De \rangle &\to \textbf{\textit{const}}\ \langle Def\ Name \rangle := \langle Stand\ Value \rangle \\
\langle var\ De \rangle &\to \langle Def\ Name \rangle : \langle Typ \rangle \\
\langle Typ \rangle &\to \textbf{\textit{int}} | \textbf{\textit{real}} | \textbf{\textit{char}} | \textbf{\textit{string}} | \textbf{\textit{boolean}} | \langle defined\ Typ \rangle \\
\langle defined\ Typ \rangle &\to \langle type\ name \rangle \\
\langle typ\ name \rangle &\to \langle Name \rangle \\
\langle type\ De \rangle &\to \textbf{\textit{type}}\ \langle Def\ Name \rangle = \langle label\ type \rangle \\
\langle label\ type \rangle &\to \langle enumeration\ De \rangle | \langle tuple\ De \rangle \\
\langle enumeration\ De \rangle &\to \textbf{\textit{enumeration of}} \langle list\ names \rangle \\
\langle tuple\ De \rangle &\to \textbf{\textit{tuple of}} \langle tuple \rangle \\
\langle tuple \rangle &\to \langle Def\ Name \rangle : \langle Typ \rangle , \langle tuple \rangle | \langle Def\ Name \rangle : \langle Typ \rangle \\
\langle Def\ Name \rangle &\to \langle Name \rangle \\
\langle Name \rangle &\to \langle Name \rangle \textbf{\textit{A}} | \langle Name \rangle \textbf{\textit{B}} | \ldots | \langle Name \rangle \textbf{\textit{z}} | \langle Name \rangle \mathbf{0} | \ldots | \langle Name \rangle \mathbf{9} \\
\langle Name \rangle &\to \textbf{\textit{A}} | \textbf{\textit{B}} | \ldots | \textbf{\textit{Z}} | \textbf{\textit{a}} | \textbf{\textit{b}} | \textbf{\textit{c}} | \ldots | \textbf{\textit{z}} \\
\langle list\ names \rangle &\to \langle Name \rangle , \langle list\ names \rangle | \langle Name \rangle \\
\langle Stand\ Value \rangle &\to \langle intStandValue \rangle | \langle realStandValue \rangle | \langle charStandValue \rangle \\
&\quad | \langle stringStandValue \rangle | \langle boolStandValue \rangle \\
\langle intStandValue \rangle &\to \langle Digit \rangle | \langle intStandValue \rangle \langle Digit \rangle \\
\langle realStandValue \rangle &\to \langle intStandValue \rangle . \langle intStandValue \rangle \\
\langle charStandValue \rangle &\to '\langle char \rangle' \\
\langle stringStandValue \rangle &\to ''\langle word \rangle'' \\
\langle boolStandValue \rangle &\to \textbf{\textit{true}} | \textbf{\textit{false}} \\
\langle Digit \rangle &\to \mathbf{0} | \mathbf{1} | \mathbf{2} | \ldots | \mathbf{9} \\
\langle char \rangle &\to \textbf{\textit{A}} | \textbf{\textit{B}} | \ldots | \textbf{\textit{Z}} | \textbf{\textit{a}} | \textbf{\textit{b}} | \ldots | \textbf{\textit{z}} | \mathbf{0} | \ldots | \mathbf{9} | + | - | \times | \textbf{/} | , | : | \ldots \\
\langle word \rangle &\to \langle word \rangle \langle char \rangle | \langle char \rangle \langle word \rangle | \epsilon
\end{aligned}
$$

C.2 Initialization Part

$$
\begin{array}{rcl}
\langle Init\rangle & \rightarrow & \langle Seq\ Instructions\rangle \\
\langle Seq\ Instructions\rangle & \rightarrow & \langle Seq\ Instruction\rangle;\langle Instruction\rangle \mid \langle Instruction\rangle \\
\langle Instruction\rangle & \rightarrow & \langle Assignation\rangle \mid \langle Cond\ Instruction\rangle \mid \langle Iter\ Instruction\rangle
\end{array}
$$

Assignation Instruction

$$
\begin{array}{rcl}
\langle Assignation\rangle & \rightarrow & \langle var\ name\rangle := \langle expression\rangle \\
\langle var\ name\rangle & \rightarrow & \langle Dec\ Name\rangle \\
\langle Dec\ Name\rangle & \rightarrow & \langle Name\rangle
\end{array}
$$

Syntax of expressions

$$
\begin{array}{rcl}
\langle Eq\ Op\rangle & \rightarrow & = \mid \neq \\
\langle Comp\ Op\rangle & \rightarrow & < \mid > \mid \leq \mid \geq \\
\langle Add\ Op\rangle & \rightarrow & + \mid - \\
\langle Mult\ Op\rangle & \rightarrow & \times \mid / \\
\langle Quant\ Op\rangle & \rightarrow & \forall \mid \exists \mid \exists_i \mid \exists_i^! \mid \textbf{\textit{choose x IN}} \mid \textbf{\textit{Max i IN}} \mid \textbf{\textit{Min i IN}} \\
\langle Log\ Op\rangle & \rightarrow & \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \\
\langle Neg\ Op\rangle & \rightarrow & \neg \\
\langle expression\rangle & \rightarrow & \langle simple\ expr\rangle\langle Eq\ Op\rangle\langle simple\ expr\rangle \\
 & & \mid \langle simple\ expr\rangle\langle Comp\ Op\rangle\langle simple\ expr\rangle \\
 & & \mid \langle simple\ expr\rangle\langle Log\ Op\rangle\langle simple\ expr\rangle \\
 & & \mid \langle Quant\ Op\rangle\langle simple\ expr\rangle \\
 & & \mid \langle Neg\ Op\rangle\langle simple\ expr\rangle \\
 & & \mid \langle simple\ expr\rangle \\
\langle simple\ expr\rangle & \rightarrow & \mid \langle simple\ expr\rangle\langle Add\ Op\rangle\langle Term\rangle \\
 & & \mid \langle simple\ expr\rangle \textbf{\textit{or}} \langle Term\rangle \\
 & & \mid \langle simple\ expr\rangle.\langle Term\rangle \\
 & & \mid \langle Term\rangle \\
\langle Term\rangle & \rightarrow & \langle Term\rangle\langle Mult\ Op\rangle\langle factor\rangle \\
 & & \mid \langle Term\rangle \textbf{\textit{and}} \langle factor\rangle \\
 & & \mid \langle factor\rangle \\
\langle factor\rangle & \rightarrow & \langle Stand\ Value\rangle \mid \langle var\ name\rangle \mid (\langle expression\rangle) \mid - \langle factor\rangle
\end{array}
$$

Conditional instructions

$$
\begin{array}{rcl}
\langle Cond\ Instruction\rangle & \rightarrow & \textbf{\textit{if}} \langle expression\rangle \textbf{\textit{then}} \langle Seq\ Instructions\rangle \textbf{\textit{else}} \langle Seq\ Instructions\rangle \textbf{\textit{fi}} \\
 & & \mid \textbf{\textit{if}} \langle expression\rangle \textbf{\textit{then}} \langle Seq\ Instructions\rangle \textbf{\textit{fi}}
\end{array}
$$

Iterative expressions

$$\langle Iterat\ Instruction\rangle \rightarrow \textbf{\textit{while}}\langle expression\rangle\textbf{\textit{do}}\langle Seq\ Instructions\rangle\textbf{\textit{od}} \mid \textbf{\textit{forall}}\langle expression\rangle\{\langle Seq\ Instructions\rangle\}$$

C.3 Synchronization and universe

$$\langle Synchro\rangle \rightarrow \textbf{\textit{Synchro\_LC}}0();|\textbf{\textit{Synchro\_LC}}1();|\textbf{\textit{Synchro\_LC}}2();$$

$$\langle Universe\rangle \rightarrow \langle AlgoName\rangle(\langle Name\rangle : \textbf{\textit{node}}) : \langle expression\rangle$$

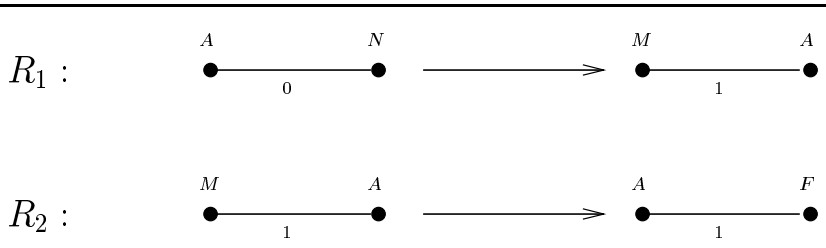
C.4 Syntax of a relabelling rule

$$\begin{aligned}
\langle Rules\rangle &\rightarrow \textbf{\textit{active rules}} : \langle Rules\ Co\rangle | \textbf{\textit{passive rules}} : \langle Rules\ Co\rangle \\
\langle Rules\ Co\rangle &\rightarrow \langle Rules\ Seq\rangle\langle expression\rangle\langle Construct\ Rules\rangle \\
\langle Construct\ Rules\rangle &\rightarrow \langle Rules\ Name\rangle := \{\ \langle RuleBody\rangle\};\langle Construct\ Rules\rangle|\epsilon \\
\langle RuleBody\rangle &\rightarrow \textbf{\textit{Precondition}}\langle precond\rangle\textbf{\textit{Relabeling}}\langle relab\rangle\textbf{\textit{Priorities}}\langle priority\rangle \\
&\quad |\textbf{\textit{Precondition}}\langle precond\rangle\textbf{\textit{Action}}\langle relab\rangle\textbf{\textit{Priorities}}\langle priority\rangle \\
\langle Rules\ Name\rangle &\rightarrow \langle Name\rangle \\
\langle precond\rangle &\rightarrow \langle expression\rangle \\
\langle relab\rangle &\rightarrow \langle expression\rangle \\
\langle priority\rangle &\rightarrow \{\langle Rules\ Seq\rangle\}; \\
\langle Rules\ Seq\rangle &\rightarrow \langle Name\rangle,\langle Rules\ Seq\rangle|\langle Name\rangle;
\end{aligned}$$

Appendix D

Catalogue of Implementation Examples

D.1 Spanning Tree: Sequential computation



Declaration
type $nLabel$ = **tuple of** var: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialiazation
$edgeLab.var := 0$;
$nodeLab.var :=$ 'N';
$G.init(nodeLab, edgeLab)$;
v : ***node***;
$v := G.choose\_node()$;
$v.label.var :=$ 'A';

Synchronization
$Synchro\_LC2()$;

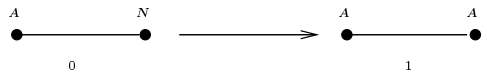
$SpanningTree(v_0$: ***node***$)$
univers
B : ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1;
$\forall w \in B(w \neq v_0,$ IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
e: ***edge***

$v_0.label.var =$ 'A'$\wedge$
$\exists w \in B(w \neq v_0 \wedge L_v[w].var =$ 'N' $\wedge \ [v_0, w].label.var = 0)$;
Relabelling
$v_0.label.var :=$ 'M';
$L_v[w].var :=$ 'A';
$[v_0, w].var := 1$;
$\forall\ v \in B(v \neq v_0,$ IMPLIES $SendTo(v, L_v[v]))$;
Priorities
{};
};
R_1:={
precondition
w: ***node***
$v_0.label.var =$ 'M'$\wedge$
$\exists w \in B(w \neq v_0 \wedge L_v[w].var =$ 'A' $\wedge \ [v_0, w].label.var = 1)$;
Relabelling
$v_0.label.var :=$ 'A';
$L_v[w].var :=$ 'F';
$\forall\ v \in B(v \neq v_0,$ IMPLIES $SendTo(v, L_v[v]))$;
Priorities
{R_0;};
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
$v : node$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $v_0.label := ReceiveFrom(v))$;
Priorities
{};
};
$SpanningTree.run(G)$;

D.2 Spanning Tree: Distributed Computation without explicit termination

R_1 :



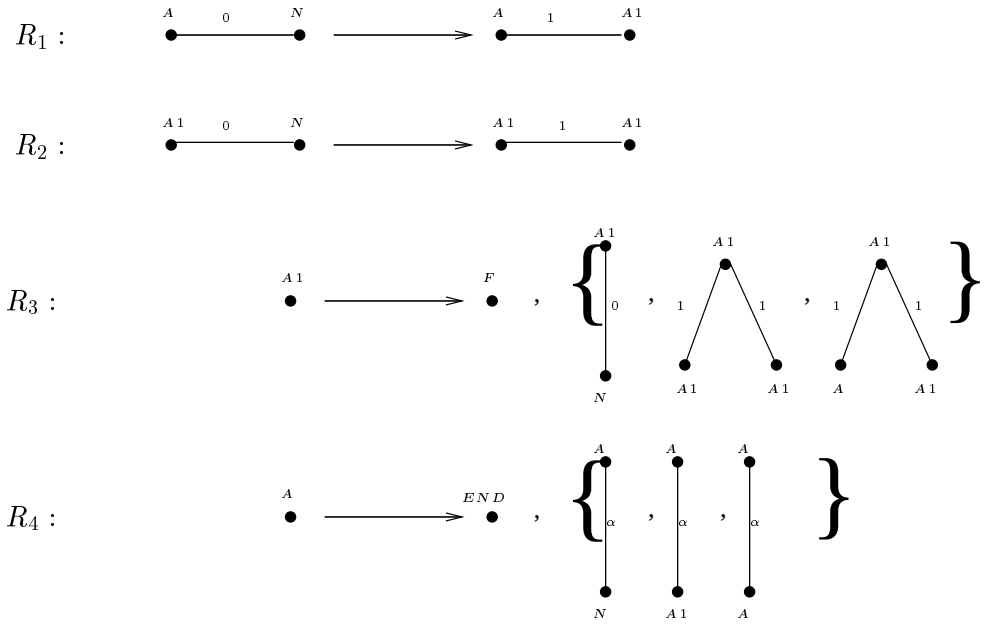
Declaration
type $nLabel$ = **tuple of** var: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$edgeLab.var = 0$;
$nodeLab.var$ = 'N';
$G.init(nodeLab, edgeLab)$;
v : ***node***;
$v := G.choose\_node()$;
$v.label.var$ = 'A';

Synchronization
$Synchro\_LC1()$;
$SpanningTree(v_0$: ***node***$)$
univers
B : ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
$v_0.label.var$ = 'N' $\wedge$
$\exists w \in B(w \neq v_0 \wedge L_v[w].var$ = 'A' $\wedge$ $[v_0, w].var = 0)$;
Relabelling
$v_0.label.var$:= 'A';
$[v_0, w].var := 1$;
Priorities
{};
};
passive rules:R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v$, IMPLIES $SendTo(v, v_0.label))$;
Priorities
{};
};
$SpanningTree.run(G)$;

D.3 Spanning Tree: Distributed Computation with explicit termination



Declaration
type $nLabel$ = **tuple of** var: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$edgeLab.var := 0$;
$nodeLab.var :=$ 'N';
$G.init(nodeLab, edgeLab)$;
v : ***node***;
$v := G.choose\_node()$;
$v.label.var :=$ 'A';

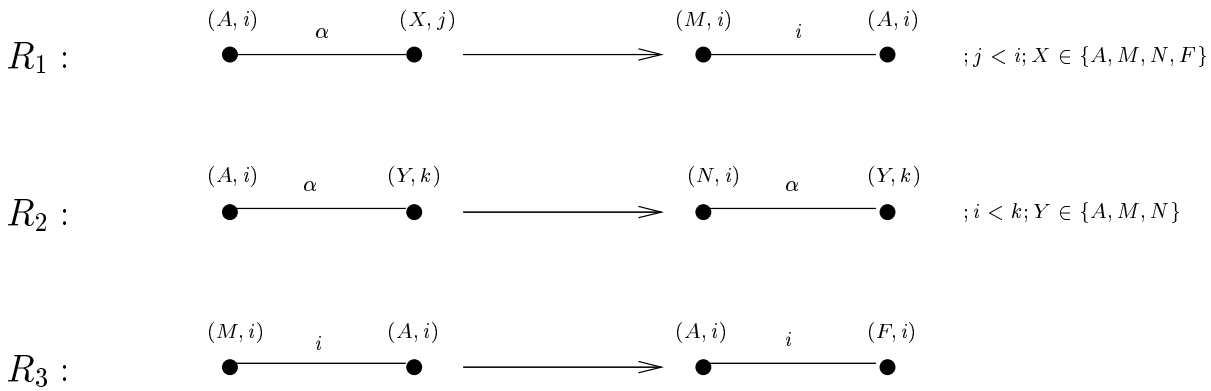
Synchronization
$Synchro\_LC1()$;
$SpanningTree(v_0$: ***node***$)$
univers
B: ***node_set***;
$B = G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1, R_2, R_3;
$\forall w \in B(w \neq v_0,$ IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
$v_0.label.var =$ 'N' $\wedge$
$\exists w \in B(w \neq v_0 \wedge L_v[w].var =$ 'A' $\wedge [v_0, w].label.var = 0)$;
Relabelling
$w.label.var :=$ 'A1';
$[v_0, w] := 1$;
Priorities
{};
};
R_1:={
precondition
w : ***node***;
$v_0.label.var =$ 'N' $\wedge$

$\exists w \in B(w \neq v_0 \wedge L_v[w].var = \text{'A1'} \wedge [v_0, w].label.var = 0)$;
Relabelling
$v_0.label.var := \text{'A1'}$;
$[v_0, w].label.var := 1$;
Priorities
{};
};
R_2:={
precondition
v_1, v_2 : ***node***;
$v_0.label.var = \text{'A1'} \wedge$
$\forall w \in B(\neg(w \neq v_0 \wedge L_v[w].var = \text{'N'} \wedge [v_0, w].label.var = 0))) \wedge$
$\forall v_1 \in B \forall v_2 \in B(\neg(v_1 \neq v_0 \wedge v_2 \neq v_0 \wedge v_1 \neq v_2 \wedge L_v[v_1].var = \text{'A1'} \wedge L_v[v_2].var = \text{'A1'} \wedge$
$[v_0, v_1].label.var = 1 \wedge [v_0, v_2].label.var = 1 \vee v_1 \neq v_0 \wedge v_2 \neq v_0 \wedge v_1 \neq v_2 \wedge L_v[v2].var = \text{'A'} \wedge$
$L_v[v2].var = \text{'A1'} \wedge [v_0, v_1].label.var = 1 \wedge [v_0, v_2].label.var = 1)$;
Relabelling
$v_0.label.var := \text{'F'}$;
Priorities
{};
};

R_3:={
precondition
v: ***node***;
$v_0.label.var = \text{'A'} \wedge \forall v \in B(L_v[v].var = \text{'F'})$;
Relabelling
$v_0.label.var := \text{'END'}$;
Priorities
{};
};
passive rules:R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall\, v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
Priorities
{};
};
$SpanningTree.run(G)$;

D.4 Spanning Tree: Sequential Computation with nodes Id.



Declaration
type $nLabel$ = **tuple of** var: **string**, id: **int**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab : eLabel$;
$nodeLab : nLabel$;

Initialization
$edgeLab.var := 0$;
$G.init(nodeLab, edgeLab)$;
v : ***node***;
i: ***int***;
$i := 0$;
$\forall v \in G$, IMPLIES $\{v.label.id := i; v.label.var :=$ 'A'$; i := i+1;\}$

Synchronization
$Synchro\_LC2()$;

$SpanningTree(v_0$: ***node***$)$
univers
B: ***node_set***;
$B = G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

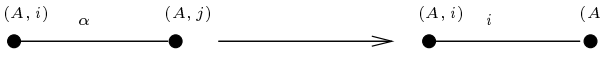
active rules: R_0, R_1, R_2;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
$v_0.label.var =$ 'A' $\wedge$ $\exists w \in B(w \neq v_0 \wedge L_v[w].id < v_0.label.id)$;
Relabelling
$L_v[w].var :=$ 'A';
$v_0.label.var :=$ 'M';
$[v_0, w].label.var := v_0.label.id$;
$\forall v \in B(v \neq v_0$, IMPLIES $SendTo(v, L_v[v]))$;
Priorities
{};
};
R_1:={
precondition
w : ***node***;
$v_0.label.var =$ 'A' $\wedge$
$\exists w \in B(w \neq v_0 \wedge L_v[w].var \neq$ 'F' $\wedge\ v_0.label.id < L_v[w].id)$;
Relabelling
$v_0.label.var :=$ 'N';
$\forall v \in B(v \neq v_0$, IMPLIES $SendTo(v, L_v[v]))$;
Priorities
{};
};
R_2:={
precondition
w : ***node***;

$v_0.label.var$ = 'M' $\wedge$
$\exists w \in B(w \neq v_0 \wedge L_v[w].var$ = 'A' $\wedge\ v_0.label.id = L_v[w].id \wedge$
$[v_0, w] = v_0.label.id)$;
Relabelling
$v_0.label.var$:= 'A';
$L_v[w].var$:= 'F';
$\forall\ v \in B(v \neq v_0,$ IMPLIES $SendTo(v, L_v[v]))$;
Priorities
$\{R_0, R_1;\}$;
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
$v : node$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $v_0.label := ReceiveFrom(v))$;
Priorities
{};
};

$SpanningTree.run(G)$;

D.5 Spanning Tree: Distributed Computation with nodes Id.

R_1 :



$; j > 0; i > 0; j < i$

Declaration
type $nLabel$ = **tuple of** var: **string**, id: **int**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$edgeLab.var = 0$;
$G.init(nodeLab, edgeLab)$;
v : ***node***;
i: ***int***;
$i := 0$;
$\forall v \in G$, IMPLIES $\{v.label.id := i; v.label.var :=$ 'A'$; i := i + 1; \}$

Synchronization
$Synchro\_LC1()$;

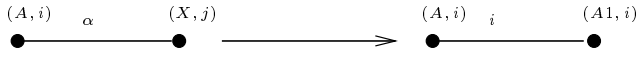
$SpanningTree(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_1;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_1:={
precondition
w: ***node***
$v_0.label.var =$ 'A' $\wedge$
$\exists w \in B(w \neq v_0 \ \wedge \ L_v[w].var =$ 'A' $\wedge L_v[w].id > v_0.label.id)$;
Relabelling
$v_0.label.id := L_v[w].id$;
$[v_0, w].label.var := L_v[w].id$;
Priorities
{};
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall v \in B(v \neq v_0 \wedge v.sync = v$, IMPLIES $SendTo(v, v_0.label))$;
Priorities
{};
};

$SpanningTree.run(G)$;

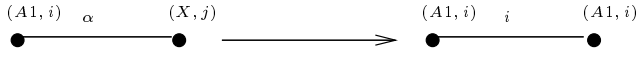
D.6 Spanning Tree: Distributed Computation with local detection of termination

R_1 :



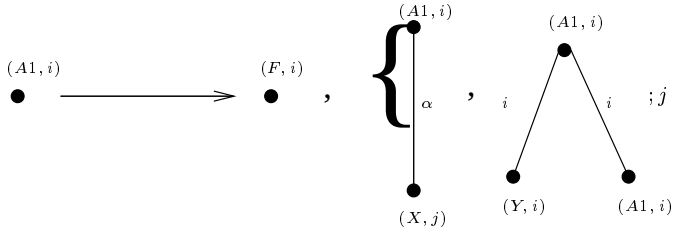
$; j < i; X \in \{A, A1, F\}$

R_2 :



$; j < i; X \in \{A, A1, F\}$

R_3 :



$; j \neq i; Y \in \{A, A1\}$

Declaration
type $nLabel$ = **tuple of** var: **string**, id: **int**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$edgeLab.var = 0$;
$G.init(nodeLab, edgeLab)$;
v : ***node***;
i: ***int***;
$i := 0$;
$\forall v \in G$, IMPLIES $\{v.label.id := i; v.label.var :=$ 'A'$; i := i + 1;\}$

Synchronization
$Synchro\_LC1()$;

$SpanningTree(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

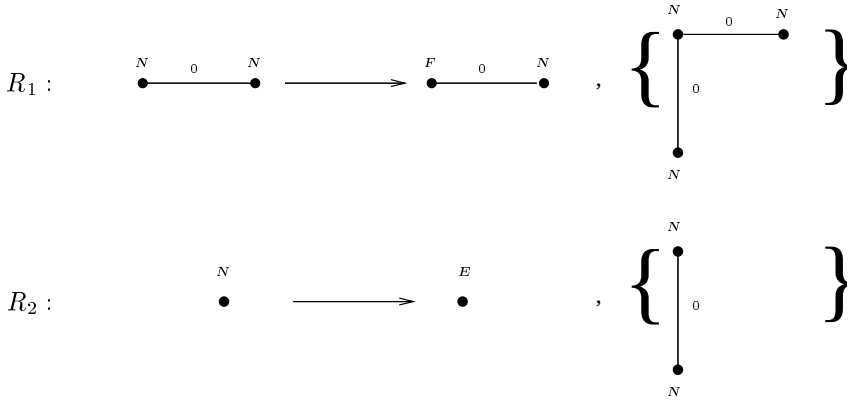
active rules: R_0, R_1, R_2;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
$\exists w \in B(w \neq v_0 \ \wedge \ L_v[w].id > v_0.label.id \wedge L_v[w].var =$ 'A'$)$;
Relabelling
$L_v[v_0].id := L_v[w].id$;
$v_0.label.var :=$ 'A1';
$[v_0, w].label.var := v_0.label.id$;
Priorities
{}
};
R_1:={
precondition
w: ***node***
$\exists w \in B(w \neq v_0 \ \wedge \ L_v[w].id > v_0.label.id \wedge L_v[w].var =$ 'A1'$)$;
Relabelling
$L_v[v_0].id := L_v[w].id$;
$v_0.label.var :=$ 'A1';
$[v_0, w].label.var := v_0.label.id$;
Priorities

{}
};
R_2:={
precondition
w: ***node***
$v_0.label.var$ = 'A1' ∧
$\forall w \in B \neg(w \neq v_0 \ \wedge \ L_v[w].id \neq v_0.label.id) \wedge$
$\forall v_1 \in B \, \forall v_2 \in B \neg(v_1 \neq v_2 \ \wedge \ v_2 \neq v_0 \ \wedge \ v_1 \neq v_0 \ \wedge \ v_0.label.id = L_v[v_1].id \ \wedge \ v_0.label.id = L_v[v_2].id \ \wedge$
$L_v[v_2].var$ = 'A1' ∧ $L_v[v_1].var \neq$ 'F');

Relabelling
$v_0.label.var$:= 'F';
Priorities
{R_1; }
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall \, v \in B(v \neq v_0 \wedge v.sync = v, \ \text{IMPLIES} \ SendTo(v, v_0.label))$;
Priorities
{};
};

$SpanningTree.run(G)$;

D.7 Election In a Tree



Declaration
type $nLabel$ = **tuple of** var: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

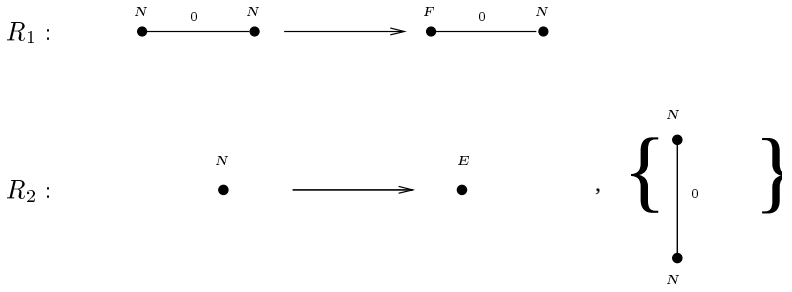
Initialization
$edgeLab.var := 0$;
$nodeLab.var :=$ 'N';
$G.init(nodeLab, edgeLab)$;
Initialization
$Synchro\_LC1()$;

$ElectionInTree(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1;
$\forall w \in B(w \neq v_0,$ IMPLIES $L_v[w] := ReceiveFrom(w))$;
$R_0 :=\{$
precondition
v : ***node***;
$v_0.label.var =$ 'N' $\wedge$ $\exists^{\perp} v \in B(v \neq v_0 \wedge L_v[v].var =$ 'N');

Relabelling
$v_0.label.var =$ 'F';
Priorities
$\{\}$;
$\};$
$R_1 :=\{$
precondition
v : ***node***
$v_0.label.var =$ 'N' $\wedge$ $\forall v \in B \neg(v \neq v_0 \wedge L_v[v].var \neq$ 'N');
Relabelling
$v_0.label.var =$ 'E';
Priorities
$\{\}$;
$\};$
passive rules: R_0;
$R_0 :=\{$
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall\, v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
Priorities
$\{\}$;
$\};$
$ElectionInTree.run(G)$;

D.8 Election in a Complete Graph



Declaration
type $nLabel$ = **tuple of** var: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

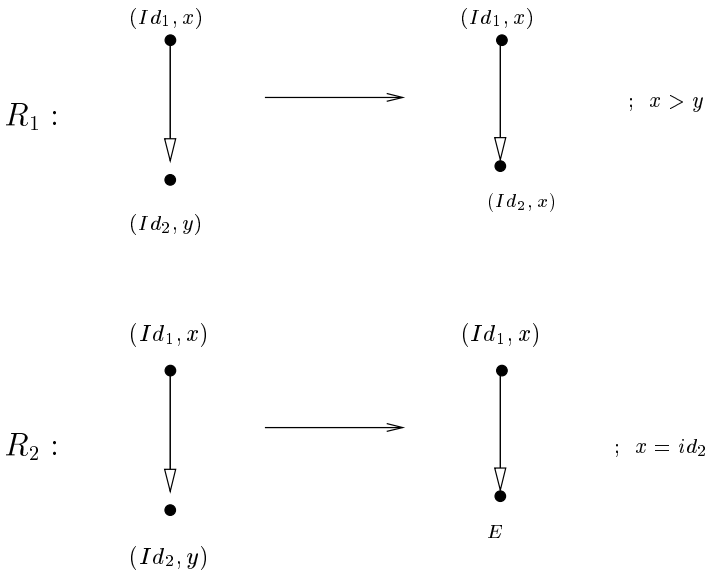
Initialization
$nodeLab.var$:= 'N';
$G.init(nodeLab, edgeLab)$;
Synchronization
$Synchro\_LC1()$;

$ElectionInGraph(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1;
$\forall w \in B(w \neq v_0,$ IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
v : ***node***
$v_0.label.var =$ 'N' $\wedge$ $\exists v \in B(v \neq v_0 \wedge L_v[v].var =$ 'N');
Relabelling
$v_0.label.var$:= 'F';
Priorities
{};
};
R_1:={
precondition
v : ***node***
$v_0.label.var =$ 'N' $\wedge$ $\forall v \in B(v \neq v_0 \wedge L_v[v].var \neq$ 'N');
Relabelling
$v_0.label.var$:= 'E';
Priorities
{};
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
Priorities
{};
};

$ElectionInGraph.run(G)$;

D.9 Election in a Ring(Chang-Robert Algorithm)



Declaration
type $nLabel$ = **tuple of** id_1: **int**, id_2: **int**,state: **string**;
type $eLabel$ = **tuple of** var: **int**;
G : ***GRAPH < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$G.init(nodeLab, edgeLab)$;
v : ***node***;
i: ***int***;
$i := 0$;
$\forall v \in G$, IMPLIES $\{L_v[v].id_1 := i; L_v[v].id_2 := i; i := i + 1;\}$

Synchronization
$Synchro\_LC1()$;

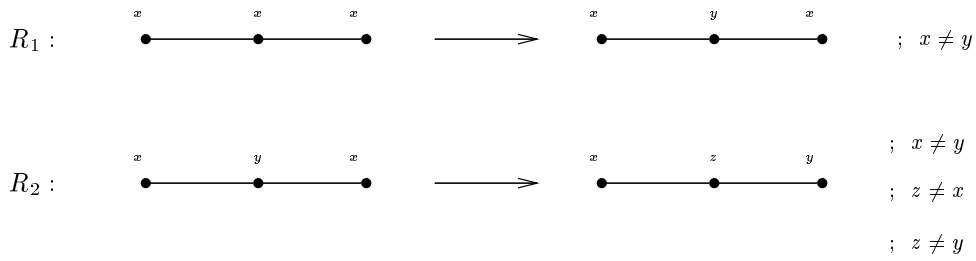
$Election(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
$\exists w \in B(w \neq v_0 \ \wedge \ L_v[w].id_2 > v_0.label.id_2)$;
Relabelling
$v_0.label.id_2 := L_v[w].id_2$;
Priorities
{}
};
R_1:={
precondition
w: ***node***
$\exists w \in B(w \neq v_0 \ \wedge \ L_v[w].id_2 = v_0.label.id_1)$;
Relabelling
$v_0.label.stat$:= 'Elected';
Priorities
{}
};
passive rules: R_0;
R_0:={
precondition

$v_0.sync \neq v_0$;
Action
$v : node$;
$\forall\ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
Priorities
{};
};

$Election.run(G)$;

D.10 3-Coloration of a Ring



Declaration
type $Color$ = **enumeration of** x,y,z;
type $nLabel$ = **tuple of** color: **Color**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$G.init(nodeLab, edgeLab)$;
v : ***node***;
$\forall v \in G$, IMPLIES $\{C$: Color; $C := Color.choose()$; $L_v[v].color := C$; $\}$

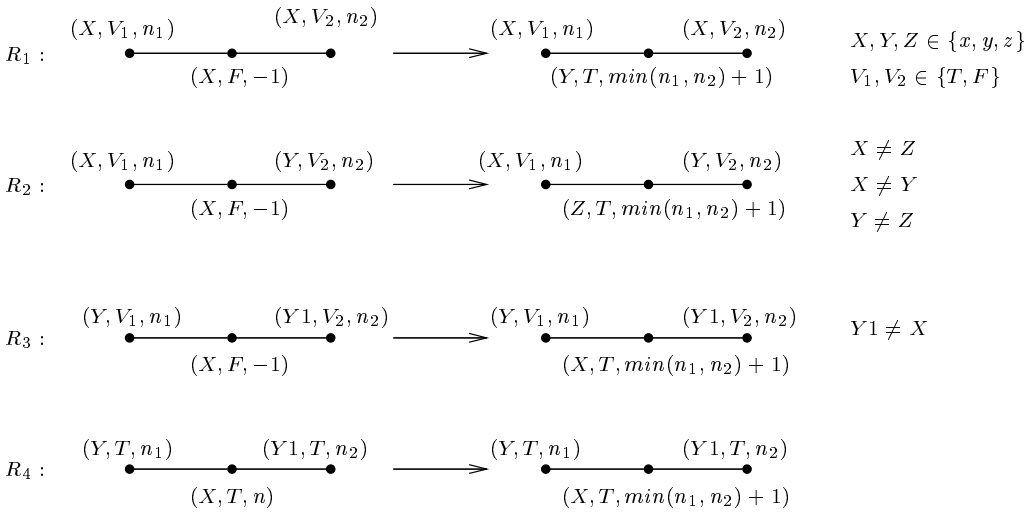
Synchronization
$Synchro\_LC1()$;

$Coloration(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
w_1: ***node***
col_1: ***Color***
$\exists w \in B \exists w_1 \in B(w \neq v_0 \ \wedge \ w_1 \neq v_0 \ \wedge \ w \neq w_1 \wedge$
$L_v[w].color = v_0.label.color \wedge$
$L_v[w_1].color = v_0.label.color \wedge$
$\exists col_1(col_1 \neq v_0.label.color)))$;
Relabelling
$v_0.label.color := col_1$;
$\forall w \in B(w \neq v_0$, IMPLIES $SendTo(w, L_v[w]))$;
Priorities
{}
};
R_1:={
precondition
w: ***node***
w_1: ***node***
col_1: ***Color***
$\exists w \in B \exists w_1 \in B(w \neq v_0 \ \wedge \ w_1 \neq v_0 \ \wedge \ w \neq w_1 \wedge$
$L_v[w].color = v_0.label.color \wedge$
$L_v[w_1].color \neq v_0.label.color \wedge$
$\exists col_1(col_1 \neq v_0.label.color \wedge$
$col_1 \neq L_v[w_1].color))$;
Relabelling
$v_0.label.color := col_1$;
Priorities
{}
};
passive rules:R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall v \in B(v \neq v_0 \wedge v.sync = v$, IMPLIES $SendTo(v, v_0.label))$;
Priorities
{};

```
};
Coloration.run(G);
```

D.11 3-Coloration + SSP Termination



Declaration
type $Color$ = **enumeration of** x,y,z;
type $nLabel$ = **tuple of** color: **Color**, lab_2: **string**, N: **int**;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

Initialization
$nodeLab.color = x$;
$nodeLab.lab_2$ = 'F';
$nodeLab.N = -1$;
$nodeLab.N = 0$;
$G.init(nodeLab, edgeLab)$;

Synchronization
$Synchro\_LC1()$;
$Coloration(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1, R_2, R_3;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w: ***node***
w_1: ***node***
$\exists w \in B\, \exists w_1 \in B(w \neq v_0 \ \wedge \ w_1 \neq v_0 \ \wedge \ w \neq w_1 \ \wedge$
$L_v[w].color = v_0.label.color \ \wedge$
$L_v[w_1].color = v_0.label.color \ \wedge$
$v_0.label.lab_2$ = 'F' $\wedge$ $v_0.label.N = -1)$;
Relabelling
$v_0.label.color := Choose\ col\ IN(\exists col \in Color(col \neq nodelabel(v_0).color))$;
$v_0.label.lab_2$:= 'T';
$v_0.label.color := min(L_v[w].N, L_v[w_1].N) + 1$;
Priorities
{}
};
R_1:={
precondition
w: ***node***
w_1: ***node***
$\exists w \in B\, \exists w_1 \in B(w \neq v_0 \ \wedge \ w_1 \neq v_0 \ \wedge \ w \neq w_1 \ \wedge$
$L_v[w].color = v_0.label.color \ \wedge$
$L_v[w_1].color \neq v_0.label.color \ \wedge$
$v_0.label.lab_2$ = 'F' $\wedge$ $v_0.label.N = -1)$;
Relabelling
$v_0.label.color := Choose\ col\ IN(\exists col \in Color(col \neq v_0.label.color \ \wedge$

$col \neq L_v[w_1].color));$
$v_0.label.lab_2 := \text{'T'};$
$v_0.label.color := min(L_v[w].N, L_v[w_1].N) + 1;$
Priorities
{}
};
R_2:={
precondition
w: ***node***
w_1: ***node***
$\exists w \in B\, \exists w_1 \in B(w \neq v_0 \ \wedge \ w_1 \neq v_0 \ \wedge \ w \neq w_1 \ \wedge$
$L_v[w_1].color \neq v_0.label.color \wedge$
$v_0.label.lab_2 = \text{'F'} \ \wedge \ v_0.label.N = -1);$
Relabelling
$v_0.label.lab_2 := \text{'T'};$
$v_0.label.color := min(L_v[w].N, L_v[w_1].N) + 1;$

Priorities
{}
};
R_3:={
precondition
w: ***node***
w_1: ***node***
$\exists w \in B\, \exists w_1 \in B(w \neq v_0 \ \wedge \ w_1 \neq v_0 \ \wedge \ w \neq w_1 \ \wedge$
$L_v[w].lab_2 = v_0.label.lab_2 \ \wedge$
$L_v[w_1].lab_2 = v_0.label.lab_2 \ \wedge \ v_0.label.lab_2 = \text{'T'});$
Relabelling
$v_0.label.color := min(L_v[w].N, L_v[w_1].N) + 1;$
Priorities
{}
};
passive rules:R_0;
R_0:={
precondition
$v_0.sync \neq v_0;$
Action
$v : node;$
$\forall\, v \in B(v \neq v_0 \wedge v.sync = v, \text{ IMPLIES } SendTo(v, v_0.label));$
Priorities
{};
};

$Coloration.run(G);$

D.12 Algorithm of Mazurkiewicz

Declaration
type $Ttuple$ = **tuple of** p_i: **int**, N_i: ***set < int >***;
type $nLabel$ = **tuple of** p: **int**, N: ***set < int >***, M: ***set < Ttuple >***;
type $eLabel$ = **tuple of** var: **int**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $eLabel$;
$nodeLab$: $nLabel$;

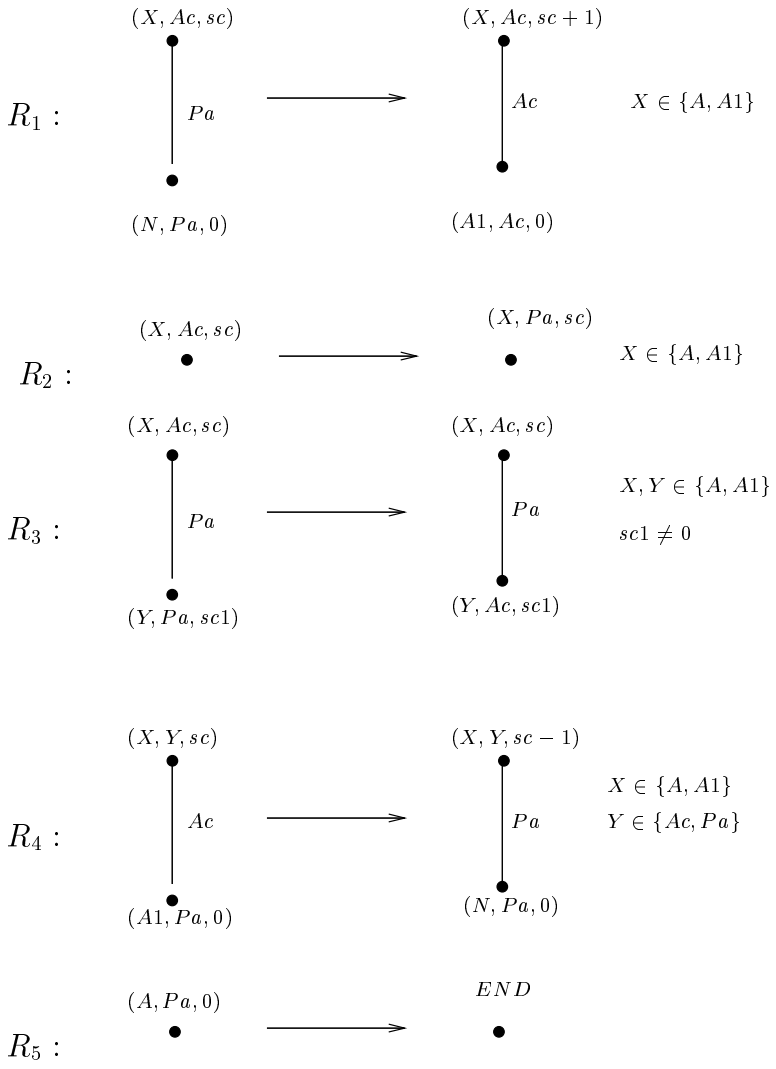
Initialization
$nodeLab.p = 0$;
$nodeLab.N = N_{init}$;
$nodeLab.M = M_{init}$;
$G.init(nodeLab, edgeLab)$;
Synchronization
$Synchro\_LC2()$;

$Mazur(v_0$: ***node***$)$
univers
B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array < nLabel >***;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1;
$\forall w \in B(w \neq v_0$, IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_1:={
precondition
w: ***node***
w_1: ***node***
$\exists w \in B \exists w_1 \in B(w \neq w_1 \ \wedge \ \forall t \in L_v[w].M(\neg L_v[w_1].M.member(t)) \vee$
$\forall t_1 \in L_v[w_1].M(\neg L_v[w].M.member(t_1)))$;
Relabelling
M: ***set < Ttuple >***;
$\forall v \in B$, IMPLIES $M := M.join(L_v[v].M)$;
$\forall v \in B$, IMPLIES $L_v[v].M := M$;
$\forall\, v \in B(v \neq v_0$, IMPLIES $SendTo(v, L_v[v]))$;
Priorities
{}
};
R_2:={
precondition
w: ***node***
w_1: ***node***
col_1: **Color**
$\forall w \in B \forall w_1 \in B(w \neq w_1 \ \wedge \ \forall t \in L_v[w].M(L_v[w_1].M.member(t)) \ \wedge$
$\forall t_1 \in L_v[w_1].M(L_v[w].M.member(t_1))) \ \wedge$
$(v_0.label.p = 0 \vee \exists t \in v_0.label.M(t.p_i = v_0.label.p \ \wedge$
$v_0.label.N.diff(t.M_i).max() < t.M_i.diff(v_0.label.N).max()))$;
Relabelling
K : **int**;
$K := Max\ i\ IN(\exists v \in B(i = L_v[v].p \vee L_v[v].N.member(i) \vee$
$\exists t \in L_v[v].M(i = t.p_i)))$;
Tt: Ttuple;
$Tt.p_i := K + 1$;
$Tt.M_i := v_0.label.N$;
$\forall v \in B$, IMPLIES $L_v[v].M.insert(Tt)$;
$\forall v \in B(v \neq v_0$, IMPLIES $L_v[v].N.delete(v_0.label.p))$;
$\forall v \in B(v \neq v_0$, IMPLIES $L_v[v].N.insert(K + 1))$;
$v_0.label.p = K + 1$;
$\forall\, v \in B(v \neq v_0$, IMPLIES $SendTo(v, L_v[v]))$;
Priorities
{}
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall\, v \in B(v \neq v_0 \wedge v.sync = v$, IMPLIES $SendTo(v, v_0.label))$;
$\forall\, v \in B(v \neq v_0 \wedge v.sync = v$, IMPLIES $v_0.label := ReceiveFrom(v))$;
Priorities
{};
};

$Mazur.run(G)$;

D.13 Algorithm of Dijkstra-Scholten



Declaration
type $nLabel$ = **tuple of** lab_1: **string**, lab_2: **string**, sc: **int**, End: **Bool**;
type $eLabel$ = **tuple of** lab: **string**;
G : ***graph < nLabel, eLabel >***;
$edgeLab$: $nLabel$;
$nodeLab$: $nLabel$;
Initialization
$nodeLab.lab_1$ = 'N';
$nodeLab.lab_2$ = 'Pa';
$nodeLab.sc = 0$;
$edgeLab.lab$ = 'Pa';
$G.init(nodeLab, edgeLab)$;
v: ***node***;
$v := G.choose\_node()$;
$v.label.lab_1$ = 'A';
$v.label.lab_2$ = 'Ac';
Synchronization
$Synchro\_LC2()$;

$Dijkstra\_Scholten(v_0 :$ ***node***$)$
univers

B: ***node_set***;
$B := G.voisinage(v_0)$;
L_v: ***node_array*** **<** ***nLabel*** **>**;
$L_v.init(B, nodeLab)$;

active rules: R_0, R_1, R_2, R_3, R_4;
$\forall w \in B(w \neq v_0,$ IMPLIES $L_v[w] := ReceiveFrom(w))$;
R_0:={
precondition
w_1: ***node***
$\neg v_0.label.End \ \wedge \ v_0.label.lab_1 \neq$ 'N' $\wedge$
$v_0.label.lab_2 =$ 'Ac' $\wedge$
$\exists w_1 \in B(L_v[w_1].lab_1 =$ 'N' $\wedge$
$L_v[w_1].lab_2 =$ 'Pa' $\wedge L_v[w_1].sc = 0 \ \wedge$
$[v_0, w_1].label.lab =$ 'Pa' $\wedge \ \neg L_v[w_1].End)$;
Relabelling
$v_0.label.sc := v_0.label.sc + 1$;
$[v_0, w_1].label.lab :=$ 'Ac';
$L_v[w_1].lab_1 :=$ 'A1';
$L_v[w_1].lab_2 :=$ 'Pa';
$\forall \ w_1 \in B(w_1 \neq v_0,$ IMPLIES $SendTo(w_1, L_v[w_1]))$;
Priorities
{}
};
R_1:={
precondition
$v_0.label.lab_1 \neq$ 'N' $\wedge \ v_0.label.lab_2 =$ 'Ac' $\wedge$
$\neg v_0.label.End$;
Relabelling
$v_0.label.lab_2 :=$ 'Pa';
$\forall \ w_1 \in B(w_1 \neq v_0,$ IMPLIES $SendTo(w_1, L_v[w_1]))$;
Priorities
{}
};
R_2:={
precondition
w_1: ***node***
$\neg v_0.label.End \ \wedge \ v_0.label.lab_1 \neq$ 'N' $\wedge$
$v_0.label.lab_2 =$ 'Ac' $\wedge$
$\exists w_1 \in B(L_v[w_1].lab_1 \neq$ 'N' $\wedge$
$L_v[w_1].lab_2 =$ 'Pa' $\wedge L_v[w_1].sc \neq 0 \ \wedge$
$[v_0, w_1].label.lab =$ 'Pa' $\wedge \ \neg L_v[w_1].End)$;
Relabelling
$L_v[w_1].lab_2 :=$ 'Ac';
$\forall \ w_1 \in B(w_1 \neq v_0,$ IMPLIES $SendTo(w_1, L_v[w_1]))$;
Priorities
{}
};
R_3:={
precondition
w_1: ***node***
$\neg v_0.label.End \ \wedge \ v_0.label.lab_1 \neq$ 'N' $\wedge$
$\exists w_1 \in B(L_v[w_1].lab_1 =$ 'A1' $\wedge L_v[w_1].lab_2 =$ 'Pa' $\wedge L_v[w_1].sc = 0 \ \wedge \ L_e[v_0, w_1].lab =$ 'Ac' $\wedge \ \neg L_v[w_1].End)$;
Relabelling
$v_0.label.sc := v_0.label.sc - 1$;
$L_v[w_1].lab_1 :=$ 'N';
$\forall \ w_1 \in B(w_1 \neq v_0,$ IMPLIES $SendTo(w_1, L_v[w_1]))$;
Priorities
{}
};
R_4:={
precondition
$v_0.label.lab_1 =$ 'A' $\wedge \ v_0.label.lab_2 =$ 'Pa' $\wedge$
$v_0.label.sc = 0 \ \wedge \ \neg v_0.label.End$;
Relabelling
$v_0.label.End := True$;
$\forall \ w_1 \in B(w_1 \neq v_0,$ IMPLIES $SendTo(w_1, L_v[w_1]))$;
Priorities
{}
};
passive rules: R_0;
R_0:={
precondition
$v_0.sync \neq v_0$;
Action
v : $node$;
$\forall \ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $SendTo(v, v_0.label))$;
$\forall \ v \in B(v \neq v_0 \wedge v.sync = v,$ IMPLIES $v_0.label := ReceiveFrom(v))$;
Priorities
{};
};

$Dijkstra\_Scholten.run(G)$;

Bibliography

[1] S. Abiteboul and V. Vianu. Generic computation and its complexity. *In proc. ACM Symp. on Theory of Computing*, New Orleans May 1991.

[2] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.

[3] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical report, LaBRI, 2002. In preparation.

[4] M. Billaud, P. Lafon, Y. Métivier, and E. Sopena. Graph rewriting systems with priorities. *Lecture notes in computer science*, 411:94–106, 1989.

[5] K. M. Chandy, , and J. Misra. *The drinking philosophers problem.* ACM Transactions on programming Languages and Systems, 1984.

[6] K. M. Chandy, , and J. Misra. *Parallel Programs Design: A Foundation.* Addison-Wesley Publishing Co., Reading, MA, April 1988.

[7] C. C. Chang and keisler H. J. *Model Theory, 3rd ed., Studies in logic.* North-Holland, 1990.

[8] P. Ciancarini, R. Gorrieri, and G. Zavattaro. An alternative semantics for the parallel operator of the calculus of gamma programs. *Imperial College Press. In Coordination Programming languages and Models and semantics.*, pages 232–248, 1996.

[9] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Operating Systems techniques, Academic Press*, 1972.

[10] M. Otto E. Grädel. Inductive definability with counting on finite structures. In *In computer science Logic, 6th Workshop, CSL 92, San Miniato 1992, Selected Papers, E. Börger, G. Jäger, H. K. Bünning, S. Martini, and M. Richter, eds., vol 702 of LNCS, Springer-Verlag*, pages 231–247, 1993.

[11] Y. Gurevich E. Grädel. Metafinite model theory. *Information and computation 140*, pages 26–81, 1998.

[12] H.-D. Ebbinghaus. *Extended logics: The general framework.* In J. Barwise and S. Feferman, editors, 1985.

[13] H.-D. Ebbinghaus and J. Flum. *Finite model theory.* Springer, 1995.

[14] Pascal Bouvry Franciszek Seredynski and Farhad Arbab. Parallel and distributed evolutionary computation with manifold., September 1997.

[15] S. Garland, N. Lynch, and M. Vaziri. Ioa: A language for specifying, programming and validating distributed systems. *Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA*, 1997.

[16] M. Grohe. Fixed-point logics on planar graphs. *In Proceedings of the 13th IEEE Symposium on Logic in Computer science*, pages 6–15, 1998.

[17] S. Grumbach and C. Tollu. On the expressive power of counting. *Theoretical computer science 149*, pages 67–99., 1995.

[18] Y. Gurevich. Logic and the challenge of computer science. *In E. Börger, editor, Current trends in theoretical computer science.*, pages 1–57, 1988.

[19] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[20] L. Hella. Logical hierarchies in ptime. *Information and computation 129*, pages 1–19., 1996.

[21] L. Hella, Ph. G. Kolaitis, and K. Luosto. Almost everywhere equivalence of logics in finite model theory. *Bulletin of symbolic Logic*, 2:422–443, 1996.

[22] L. Hella, L. Libkin, and J. Nurmonen. Notions of locality and their logical characterizations over finite models. *Journal of symbolic logic 64*, pages 1751–1773., 1999.

[23] C. A. R. Hoare. *Communicating Sequential processes.* Prentice-Hall International, United Kingdom, 1985.

[24] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems, 16(3)*, pages 872–923, May 1989.

[25] S. Lindell. An analysis of fixed-point queries on binary trees. *Theoretical computer science*, 85:75–95, 1991.

[26] N. A. Lynch. *Distributed algorithms.* Morgan Kaufman, 1996.

[27] J. Marino M. Grohe. Definability and descriptive complexity on databases of bounded tree-width. In *In C. Beeri. proceedings of the 7th international Conference on Database Theory, lecture notes in computer science. Springer Verlag*, 1999.

[28] Y. Métivier, N. Saheb, and A. Zemmari. Randomized rendezvous. In *Mathematics and computer science : Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194. Birkhäuser, 2000.

[29] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Inform. Proc. Letters*, pages 82:313–120, 2002.

[30] Y. Métivier and G. Tel. Termination detection and universal graph reconstruction. In *International Colloquium on structural information and communication complexity*, pages 237–251. Carleton scientific press, 2000.

[31] R. Milner. *Communication and Concurrency.* Prentice-Hall International, United Kingdom, 1989.

[32] M. Mosbah and R. Ossamy. A programming language for local computations in graphs: Logical basis. Technical report, LaBRI, 2004. In preparation.

[33] M. Mosbah and A. Sellami. Visidia: A tool for the visualization and similation of distributed algorithms. *http://www.labri.fr/visidia/.*

[34] M. Mosbah, A. Sellami, and A. Zemmari. using graph relabelling systems for resolving conflicts. 2004.

[35] MPI. A message-passing interface standard version 1.1. message passing interface forum, university of tenessee, knoxville, June, 1995. *http://www.mcs.anl.gov/Projects/mpi/mpich/index.html.*

[36] E. Lander N. Immerman. Describing graphs: A first-order approach to graph canonization. *In A. Selman, editor, Complexity theory restrospective*, pages 59–81, 1990.

[37] W. Weihl N. Lynch, M. Merritt and A. Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[38] M. Otto. The expressive power of fixed-point logic with counting. *Journal of symbolic Logic*, 61:147–176, 1996.

[39] M. Otto. *Bounded Variable logics and Counting.* Springer-Verlag, 1997.

[40] E. RUTTEN, F. ARBAB, and I. HERMAN. Formal specification of manifold: A preliminary study. *Tech. Rep. CS-R9215, Centrum voor Wiskunde en Informatica, Kruislan 413,1098 SJ Amsterdam The Netherlands,*, 1992.

[41] E. RUTTEN and Minifold. A kernel for the coordination language manifold. *Tech. Rep. CS-R9252, Centrum voor Wiskunde en Informatica, Amsterdam,*, November 1992.

[42] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192:315–351, 1998.

[43] S. Shelah Y. Gurevich. Fixed-point extensions of firdt-order logic. *Annals of Pure and Applied Logic 32*, pages 265–280, 1986.

[44] B. Szymanski Y. Shi and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. In *4th International Conference on Distributed Computing Systems*, pages 287–292, 1985.