Towards synthesis of distributed algorithms with SMT solvers*

Carole Delporte-Gallet, Hugues Fauconnier, Yan Jurski, François Laroussinie, and Arnaud Sangnier

IRIF, Univ Paris Diderot, CNRS, France

Abstract. We consider the problem of synthesizing distributed algorithms working on a specific execution context. We show it is possible to use the linear time temporal logic in order to both specify the correctness of algorithms and their execution contexts. We then provide a method allowing to reduce the synthesis problem of finite state algorithms to some model-checking problems. We finally apply our technique to automatically generate algorithms for consensus and epsilon-agreement in the case of two processes using the SMT solver Z3.

Introduction

On the difficulty to design correct distributed algorithms. When designing distributed algorithms, researchers have to deal with two main problems. First, it is not always possible to find an algorithm which solves a specific task. For instance, it is known that there is no algorithm for distributed consensus in the full general case where processes are subject to failure and communication is asynchronous[6]. Second, they have to prove that their algorithms are correct, which can sometimes be very tedious due to the number of possible executions to consider. Moreover distributed algorithms are often designed by assuming a certain number of hypothesis which are sometimes difficult to properly formalize.

Even though most distributed algorithms for problems like leader election, consensus, set agreement, or renaming, are not very long, their behavior is difficult to understand due to the numerous possible interleavings and their correctness proofs are extremely intricate. Furthermore these proofs strongly depend on the specific assumptions made on the *execution context* which specifies the way the different processes are scheduled and when it is required for a process to terminate. In the case of distributed algorithms with shared registers, interesting execution contexts are for instance the *wait-free* model which requires that each process terminates after a finite number of its own steps, no matter what the other processes are doing [8] or the *obstruction-free* model where every process that eventually executes in isolation has to terminate [9]. It is not an easy task to describe formally such execution context and the difference between contexts can be crucial when searching for a corresponding distributed algorithm. As a matter of fact, there is no wait-free distributed algorithm to solve consensus [10], even with only two processes, but there exist algorithms in the obstruction-free case.

* Supported by ANR FREDDA (ANR-17-CE40-0013).

Proving correctness vs synthesis. When one has designed a distributed algorithm for a specific execution context, it remains to prove that it behaves correctly. The most common way consists in providing a 'manual' proof hoping that it covers all the possible cases. The drawback of this method is that manual proofs are subject to bugs and they are sometimes long and difficult to check. It is often the case that the algorithms and their specification are described at a high-level point of view which may introduce some ambiguities in the expected behaviors. Another approach consists in using automatic or partly automatic techniques based on formal methods. For instance, the tool TLA+ [3] provides a language to write proofs of correctness which can be checked automatically thanks to a proof system. This approach is much safer, however finding the correct proof arguments so that the proof system terminates might be hard. For finite state distributed algorithms, another way is to rely on model-checking [2, 14]. Here, a model for the algorithm together with a formula specifying its correctness, expressed for example in temporal logics like LTL or CTL [5], are given, and checking whether the model satisfies the specification is then automatic. This is the approach of the tool SPIN [11] which has allowed to verify many algorithms.

These methods are useful when they succeed in showing that a distributed algorithm is correct, but when it appears that the algorithm does not respect its specification, then a new algorithm has to be conceived and the tedious work begins again. One way to solve this issue is to design distributed algorithms which are correct by construction. In other words, one provides a specification and then an automatic tool synthesizes an algorithm for this specification. Synthesis has been successfully applied to various kinds of systems, in particular to design reactive systems which have to take decisions according to their environment: in such cases, the synthesis problem consists in finding a winning strategy in a two player games (see for instance [7]). In a context of distributed algorithms, some recent works have developed some synthesis techniques in order to obtain automatically some thresholds bounds for fault-tolerant distributed algorithms [12]. The advantage of such methods is that the synthesis algorithm can be used to produce many distributed algorithms and there is no need to prove that they are correct, the correctness being ensured (automatically) by construction.

Our contributions. In this work, we first define a simple model to describe distributed algorithms for a finite number of processes communicating thanks to shared registers. We then show that the correctness of these algorithms can be specified by a formula of the linear time temporal logic LTL[13, 15] and more interestingly we show that classical execution contexts can also be specified in LTL. We then provide a way to synthesize automatically distributed algorithms from a specification. Following SAT-based model-checking approach [1], we have furthermore implemented our method in a prototype which relies on the SMT-solver Z3 [4] and for some specific cases synthesizes non-trivial algorithms. Of course the complexity is high and we can at present only generate algorithms for two processes but they are interesting by themselves and meet their specification w.r.t. several execution contexts.

1 Distributed algorithms and specification language

1.1 Distributed algorithms with shared memory

We begin by defining a model to represent distributed algorithms using shared memory. In our model, each process is equipped with an atomic register that it is the only one to write but that can be read by all the others processes (*single writer-multiple readers registers*).

The processes manipulate a data set $\mathcal{D}$ including a set of input values $\mathcal{D}_\mathcal{I} \subseteq \mathcal{D}$, a set of output values $\mathcal{D}_\mathcal{O} \subseteq \mathcal{D}$ and a special value $\bot \in \mathcal{D} \setminus (\mathcal{D}_\mathcal{I} \cup \mathcal{D}_\mathcal{O})$ used to characterize a register that has not yet been written. The actions performed by the processes are of three types, they can either write a data in their register, read the register of another process or decide a value. For a finite number of processes n, we denote by $Act(\mathcal{D}, n) = \{\mathbf{wr}(d), \mathbf{re}(k), \mathbf{dec}(o) \mid d \in \mathcal{D} \setminus \{\bot\}, k \in [1, n], o \in \mathcal{D}_\mathcal{O}\}$ where $\mathbf{wr}(d)$ stands for "write the value d to the register", $\mathbf{re}(k)$ for "read the register of process k", and $\mathbf{dec}(o)$ for "output (or decide) the value o".

The action performed by a process at a specific instant depends on the values it has read in the registers of the other processes, we hence suppose that each process stores a local copy of the shared registers that it modifies when it performs a read or a write. Furthermore, in some cases, a process might perform different actions with the same local copy of the registers, because for instance it has stored some information on what has happened previously. This is the reason why we equip each process with a local memory as well. A process looking at its copy of the registers and at its memory value decides to perform an unique action on its local view and to update its memory. According to this, we define the code executed by a process in a distributed algorithm as follows.

Definition 1 (Process algorithm). *A* process algorithm P *for an environment of n processes over the data set $\mathcal{D}$ is a tuple (M, δ) where:*

1. *M is a finite set corresponding to the local memory values of the process;*
2. *$\delta : \mathcal{D}_\mathcal{I} \cup (\mathcal{D}^n \times M) \mapsto Act(\mathcal{D}, n) \times M$ is the action function which determines the next action to be performed and the update of the local memory, such that if $\delta(s) = (\mathbf{dec}(o), m')$ then $s = (\mathbf{V}, m) \in \mathcal{D}^n \times M$ and $m = m'$.*

A pair $(a, m) \in Act(\mathcal{D}, n) \times M$ is called a *move*. The last condition ensures that a process first move cannot be to decide a value (this is only to ease some definitions) and when a process has decided then it cannot do anything else and its decision remains the same. Note that the first move to be performed by the process from an input value i in $\mathcal{D}_\mathcal{I}$ is given by $\delta(i)$.

A process state s for a process algorithm P is either an initial value in $\mathcal{D}_\mathcal{I}$ or a pair $(\mathbf{V}, m) \in \mathcal{D}^n \times M$ where the first component corresponds to the local view of the processes and m is the memory value. Let $\mathcal{S}_P \subseteq \mathcal{D}_\mathcal{I} \cup (\mathcal{D}^n \times M)$ the states associated to P. An initial state belongs to $\mathcal{D}_\mathcal{I}$. We now define the behavior of a process when it has access to a shared memory $\mathbf{R} \in \mathcal{D}^n$ and its identifier in the system is $i \in [1, n]$. For this we define a transition relation $\xrightarrow{i} \subseteq (\mathcal{S}_P \times \mathcal{D}^n) \times (Act(\mathcal{D}, n) \times M) \times (\mathcal{S}_P \times \mathcal{D}^n)$ such that $(s, \mathbf{R}) \xrightarrow{i,(a,m')} (s', \mathbf{R}')$

iff for all $j \in [1,n]$ if $i \neq j$ then $\mathbf{R}[j] = \mathbf{R}'[j]$, and we are in one of the the following cases:

1. if $a = \mathbf{wr}(d)$ then $\mathbf{R}'[i] = d$ and $s' = (\mathbf{V}', m')$ such that $\mathbf{V}'[i] = d$ and, for all $j \in [1,n] \setminus \{i\}$, if $s = (\mathbf{V}, m)$ (i.e. $s \notin \mathcal{D}_{\mathcal{I}}$) then $\mathbf{V}'[j] = \mathbf{V}[j]$ and otherwise $\mathbf{V}'[j] = \bot$ i.e. the write action updates the corresponding shared register as well as the local view.
2. if $a = \mathbf{re}(k)$ then $\mathbf{R}' = \mathbf{R}$, and $s' = (\mathbf{V}', m')$ (i.e. $s \notin \mathcal{D}_{\mathcal{I}}$) with $\mathbf{V}'[k] = \mathbf{R}[k]$ and, for all $j \in [1,n] \setminus \{k\}$, if $s = (\mathbf{V}, m)$ then $\mathbf{V}'[j] = \mathbf{V}[j]$ and otherwise $\mathbf{V}'[j] = \bot$, i.e. the read action copies the value of the shared register of process k in the local view.
3. if $a = \mathbf{dec}(o)$ then $\mathbf{R}' = \mathbf{R}$ and $s' = s$, i.e. the decide action does not change the local state of any process, neither the shared registers.

The transition relation $\xrightarrow{i}_P \subseteq (\mathcal{S}_P \times \mathcal{D}^n) \times (\mathcal{S}_P \times \mathcal{D}^n)$ associated to the process algorithm P is defined by: $(s, \mathbf{R}) \xrightarrow{i}_P (s', \mathbf{R}')$ iff $(s, \mathbf{R}) \xrightarrow{i,\delta(s)} (s', \mathbf{R}')$. Different process algorithms can then be combined to form a distributed algorithm.

Definition 2 (Distributed algorithm). *A n processes* distributed algorithm *A over the data set $\mathcal{D}$ is given by $P_1 \otimes P_2 \otimes \ldots \otimes P_n$ where P_i is a process algorithm for an environment of n processes over the data set $\mathcal{D}$ for all $i \in [1,n]$.*

We now define the behavior of such a n processes distributed algorithm $P_1 \otimes P_2 \otimes \ldots \otimes P_n$. We call a *configuration* of A a pair of vectors $C = (\mathbf{S}, \mathbf{R})$ where $\mathbf{S}$ is a n dimensional vector such that $\mathbf{S}[i] \in \mathcal{S}_{P_i}$ represents the state for process i and $\mathbf{R} \in \mathcal{D}^n$ represents the values of the shared registers. We use $\mathcal{C}_A$ to represent the set of configurations of A. The initial configuration for the vector of input values $\mathbf{In} \in \mathcal{D}^n_{\mathcal{I}}$ is then simply $(\mathbf{In}, \mathbf{R})$ with $\mathbf{R}[i] = \bot$ for all $i \in [1,n]$. Given a process identifier $i \in [1,n]$ and a pair (a, m) where $a \in Act(\mathcal{D}, n)$ and m is a memory value for process i, we define the transition relations $\xRightarrow{i,(a,m)}$ over configurations as $(\mathbf{S}, \mathbf{R}) \xRightarrow{i,(a,m)} (\mathbf{S}', \mathbf{R}')$ iff we have $(\mathbf{S}[i], \mathbf{R}) \xrightarrow{i,(a,m)} (\mathbf{S}'[i], \mathbf{R}')$ and for every $j \neq i$: $\mathbf{S}'[j] = \mathbf{S}[j]$. The execution step $\xRightarrow{i}_A$ of process i for the distributed algorithm A is defined by $(\mathbf{S}, \mathbf{R}) \xRightarrow{i}_A (\mathbf{S}', \mathbf{R}')$ iff $(\mathbf{S}[i], \mathbf{R}) \xrightarrow{i}_{P_i} (\mathbf{S}'[i], \mathbf{R}')$, note that in that case we have $(\mathbf{S}, \mathbf{R}) \xRightarrow{i,\delta_i(S[i])} (\mathbf{S}', \mathbf{R}')$ if δ_i is the action function of P_i.

1.2 Example

Algorithm 1 provides a classical representation of a tentative distributed algorithm to solve consensus with two processes. Each process starts with an input value V and the consensus goal is that both processes eventually decide the same value which must be one of the initial values. It is well known that there is no wait-free algorithm to solve consensus [6, 8] hence this algorithm will not work for any set of executions, in particular one could check that if the two processes start with a different input value and if they are executed in a round-robin manner (i.e. process 1 does one step and then process 2 does one and so on) then none of the

Algorithm 1 Consensus algorithm for process i with i $\in \{1, 2\}$

Require: V: the input value of process i
1: **while** true **do**
2: r[i]:=V
3: tmp:=r[3-i]
4: **if** tmp=V or tmp = $\perp$ **then**
5: Decide(V)
6: Exit()
7: **else**
8: V:=tmp
9: **end if**
10: **end while**

process will ever decide and they will exchange their value for ever. We shall see however later that under some restrictions on the set of considered executions this algorithm solves consensus.

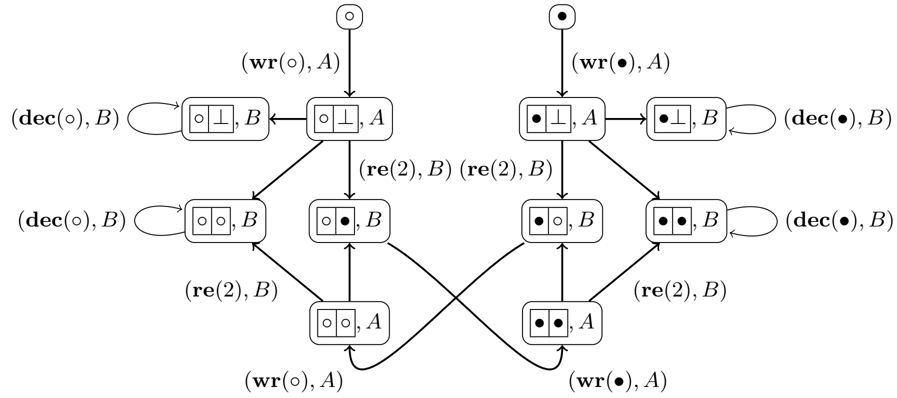

Fig. 1. View of a process algorithm P for a process with identifier 1

Figure 1 gives a visual description of the process algorithm corresponding to the Algorithm 1 supposing that the corresponding process has identifier 1. In this graph, each nodes represents a process state, the memory is the set $\{A, B\}$ and the data belongs to $\{\circ, \bullet\}$. From each node, we have some edges labeled with the action to perform according to the process state. The first action consists in writing the input data in the register, which leads to a state where the local view contains the data in the first register and $\perp$ in the local copy of the second register and the local memory cell is A. Afterwards, the process reads the second register and on Figure 1, we represent all the possible data that could be in this register (i.e either $\circ$, $\bullet$ or $\perp$) in the local view and the memory cell evolves to B.

Hence, the elements A and B of the memory set are used to represent the local state of the algorithm: when the local memory is A it means that the last action performed by the process was the write action corresponding to the Line 2 of Algorithm 1 and when its value is B, it means that the Algorithm has performed the read action corresponding to the Line 3. We only need these two values for the memory, because in our setting after having read the memory, the read value is stored in the local copy of the register and according to it, the algorithm either decides or goes back to Line 2. Note that when we leave one of the state at the bottom of the figure by reading the second register, we take into account that $\perp$ cannot be present in this register, since at this stage this register has necessarily been written.

2 Using *LTL* to reason on distributed algorithms

2.1 Kripke structures and *LTL*

We specify distributed algorithms with the Linear time Temporal Logic (LTL). We recall here some basic definitions concerning this logic and how its formulae are evaluated over Kripke structures labeled with atomic propositions from a set AP.

Definition 3 (Kripke structure). *A* Kripke structure $\mathcal{K}$ *is a* 4*-tuple* (Q, E, ℓ, q_{init}) *where Q is a countable set of states, $q_{init} \in Q$ is the initial state, $E \subseteq Q^2$ is a total* [1] *relation and $\ell \colon Q \to 2^{\mathsf{AP}}$ is a labelling function.*

A path (or an execution) in $\mathcal{K}$ from a state q is an infinite sequence $q_0 q_1 q_2 \cdots$ such that $q_0 = q$ and $(q_i, q_{i+1}) \in E$ for any i. We use $\mathsf{Path}_{\mathcal{K}}(q)$ to denote the set of paths from q. Given a path ρ and $i \in \mathbb{N}$, we write ρ^i for the path $q_i q_{i+1} q_{i+2} \ldots$ (the i-th suffix of ρ) and $\rho(i)$ for the i-th state q_i.

In order to specify properties over the execution of a Kripke structure, we use the Linear time Temporal Logic (LTL) whose syntax is given by the following grammar $\phi, \psi ::= p \mid \neg\phi \mid \phi \vee \psi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \psi$ where p ranges over AP. We use standard abbreviations: $\top$, $\perp$, $\vee$, $\Rightarrow$... as well as the classical temporal modalities $\mathbf{F}\phi \stackrel{\text{def}}{=} \top \mathbf{U} \phi$ and $\mathbf{G}\phi \stackrel{\text{def}}{=} \neg \mathbf{F} \neg \phi$. Given a path ρ of a Kripke structure $\mathcal{K} = (Q, E, \ell, q_{\text{init}})$, the satisfaction relation $\models$ for LTL is defined inductively by:

$$\begin{aligned}
\rho \models p &\text{ iff } p \in \ell(\rho(0)) \\
\rho \models \neg\phi &\text{ iff } \rho \not\models \phi \\
\rho \models \phi \vee \psi &\text{ iff } \rho \models \phi \text{ or } \rho \models \psi \\
\rho \models \mathbf{X}\phi &\text{ iff } \rho^1 \models \phi \\
\rho \models \phi \mathbf{U} \psi &\text{ iff } \exists i \geq 0.\ \rho^i \models \psi \text{ and } \forall 0 \leq j < i.\ \rho^j \models \phi
\end{aligned}$$

We then write $\mathcal{K} \models \phi$ iff $\rho \models \phi$ for any $\rho \in \mathsf{Path}_{\mathcal{K}}(q_{\text{init}})$. Since we quantify over all the paths, we speak of universal model-checking.

[1] *I.e.*, for all $q \in Q$, there exists $q' \in Q$ s.t. $(q, q') \in E$.

2.2 Specifying distributed algorithms

We will now see how to use LTL formulae for specifying the correctness of distributed algorithms under specific execution contexts. We consider distributed algorithms for n processes working over a data set $\mathcal{D}$. The set of atomic propositions that we will use in this context will then be : $\mathsf{AP}^n_{\mathcal{D}} = \{\mathsf{active}_i, \mathsf{D}_i\}_{1 \leq i \leq n} \cup \{\mathsf{In}^d_i\}_{1 \leq i \leq n, d \in \mathcal{D}_\mathcal{I}} \cup \{\mathsf{Out}^d_i\}_{1 \leq i \leq n, d \in \mathcal{D}_\mathcal{O}}$ where active_i represents the fact that process i has been the last one to execute an action, D_i that process i has decided, In^d_i that the initial value of process i is d and Out^d_i that the output value of process i is d. Note that we always have: $\mathsf{D}_i \Leftrightarrow \bigvee_d \mathsf{Out}^d_i$.

We shall now see how we associate a Kripke structure labeled with these propositions with a distributed algorithm. Let $A = P_1 \otimes P_2 \otimes \ldots \otimes P_n$ be a n process distributed algorithm over the data set $\mathcal{D}$. The states of the Kripke structures contain configurations of A together with information on which was the last process to perform an action as well as the output value for each process (set to $\perp$ if the process did not output any value yet). Formally, we define $\mathcal{K}_A = (Q_A, E_A, \ell_A, q^A_{\text{init}})$ with:

- $Q_A = \{q^A_{\text{init}}\} \cup (\mathcal{C}_A \times [0, n] \times (\mathcal{D}_\mathcal{O} \cup \{\perp\})^n)$, the first component is a configuration of A, the second is the identifier of the last process which has performed an action (it is set to 0 at the beginning), the third contains the output value;
- E_A is such that:
  - $\big(q^A_{\text{init}}, ((\mathbf{In}, \perp), 0, \perp)\big) \in E$ for all initial configurations $(\mathbf{In}, \perp)$ of A (here $\perp$ stands for the unique vector in $\{\perp\}^n$), i.e. the initial configurations are the one accessible from the initial state q_{init} after one step,
  - $\big(((\mathbf{S}, \mathbf{R}), i, \mathbf{O}), ((\mathbf{S}', \mathbf{R}'), j, \mathbf{O}')\big) \in E_A$ iff $(\mathbf{S}, \mathbf{R}) \overset{j}{\Rightarrow}_A (\mathbf{S}', \mathbf{R}')$ and if the action performed by process j (from $\mathbf{S}[j]$ to $\mathbf{S}'[j]$) is $\mathbf{dec}(o)$ then $\mathbf{O}'[j] = o$ and $\mathbf{O}'[k] = \mathbf{O}[k]$ for all $k \in [1, n] \setminus \{j\}$, otherwise $\mathbf{O} = \mathbf{O}'$.
- the labelling function ℓ_A is such that:
  - $\ell_A(q^A_{\text{init}}) = \emptyset$,
  - $\mathsf{active}_i \in \ell_A((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ and $\mathsf{active}_j \notin \ell((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ if $j \neq i$, i.e the last process which has performed an action is i,
  - $\mathsf{In}^d_j \in \ell_A((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ iff $\mathbf{S}[j] \in \mathcal{D}_\mathcal{I}$ and $d = \mathbf{S}[j]$, i.e. process j is still in its initial configuration with its initial value d,
  - $\mathsf{D}_j \in \ell_A((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ iff $\mathbf{O}[j] \neq \perp$, i.e. process j has output its final value;
  - Out^d_j iff $\mathbf{O}[j] = d$, i.e. the value output by process j is d.

For a LTL formula ϕ over $\mathsf{AP}^n_{\mathcal{D}}$, we say that the distributed algorithm A satisfies ϕ, denoted by $A \models \phi$, iff $\mathcal{K}_A \models \phi$.

The LTL formulae over $\{\mathsf{In}^d_i\}_{1 \leq i \leq n, d \in \mathcal{D}_\mathcal{I}} \cup \{\mathsf{Out}^d_i\}_{1 \leq i \leq n, d \in \mathcal{D}_\mathcal{O}}$ will be typically used to state some correctness properties about the link between input and output values. The strength of our specification language is that it allows to specify execution contexts thanks to the atomic propositions in $\{\mathsf{active}_i, \mathsf{D}_i\}_{1 \leq i \leq n}$.

Even if this is not the main goal of this research work, we know that given a n processes distributed algorithm A over a finite data set $\mathcal{D}$ and a LTL formula Φ over $\mathsf{AP}^n_{\mathcal{D}}$, one can automatically verify whether $A \models \Phi$ and this can be done in polynomial space. Indeed model-checking an LTL formula Φ over a Kripke

structure can be achieved in polynomial space [15]: the classical way consists in using a Büchi automaton corresponding to the negation of the formula Φ (which can be of exponential size in the size of the formula) and then checking for intersection emptiness *on the fly* (the automaton is not built but traveled). The same technique can be applied here to verify $A \models \Phi$ without building explicitly $\mathcal{K}_A$. Therefore we have the following result which is a direct consequence of [15]:

Proposition 1. *Given a n processes distributed algorithm A over a finite data set $\mathcal{D}$ and a LTL formula Φ over $\mathsf{AP}^n_{\mathcal{D}}$, verifying whether $A \models \Phi$ is in* PSPACE.

2.3 Examples

Specification for consensus algorithms. We recall that the consensus problem for n processes can be stated as follows: each process is equipped with an initial value and then all the processes that decide must decide the same value (*agreement*) and this value must be one of the initial one (*validity*). We do not introduce for the moment any constraints on which process has to propose an output, this will come later. We assume that the consensus algorithms work over a data set $\mathcal{D}$ with $\mathcal{D}_{\mathcal{I}} = \mathcal{D}_{\mathcal{O}}$, i.e. the set of input values and of output values are equal. The agreement can be specified by the following formula:

$$\Phi^c_{\text{agree}} \stackrel{\text{def}}{=} \mathbf{G} \bigwedge_{1 \leq i \neq j \leq n} \Big((\mathsf{D}_i \wedge \mathsf{D}_j) \Rightarrow (\bigwedge_{d \in \mathcal{D}_{\mathcal{O}}} \mathsf{Out}^d_i \Leftrightarrow \mathsf{Out}^d_j) \Big)$$

We state here that if two processes have decided a value, then this value is the same. For what concerns the validity, it can be expressed by:

$$\Phi^c_{\text{valid}} \stackrel{\text{def}}{=} \mathbf{X} \bigwedge_{1 \leq i \leq n} \bigwedge_{d \in \mathcal{D}_{\mathcal{I}}} \Big(\Big(\mathbf{F}\, \mathsf{Out}^d_i \Big) \Rightarrow \Big(\bigvee_{1 \leq j \leq n} \mathsf{In}^d_j \Big) \Big)$$

In this case, the formula simply states that if eventually a value is output, then this value was the initial value of one the processes. Note that this formula begins with the temporal operator $\mathbf{X}$ because in the considered Kripke structure the initial configurations are reachable after one step from q_{init}.

We are now ready to provide specifications for the execution context, i.e. the formulae which tell when processes have to decide. First we consider a *wait-free* execution context, each process produces an output value after a finite number of its own steps, independently of the steps of the other processes [8]. This can be described by the *LTL* formula:

$$\Phi_{\text{wf}} \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq n} \Big((\mathbf{G}\,\mathbf{F}\, \mathsf{active}_i) \Rightarrow (\mathbf{F}\, \mathsf{D}_i) \Big)$$

This formula states that for each process, if it is regularly (infinitely often) active, then at some point (i.e. after a finite number of steps) it must decide. Consequently if a distributed algorithm A is such that $A \models \Phi^c_{\text{agree}} \wedge \Phi^c_{\text{valid}} \wedge \Phi_{\text{wf}}$, then A is a wait-free distributed algorithm for consensus. However we know that even for two

processes such an algorithm does not exist [6, 8]. But, when considering other execution contexts, it possible to have an algorithm for consensus.

An another interesting execution context is the *obstruction-free* context. Here, every process that eventually executes in isolation has to produce an output value [9]. This can be ensured by the following LTL formula which exactly matches the informal definition.

$$\Phi_{\text{of}} \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq n} ((\mathbf{F}\,\mathbf{G}\,\mathsf{active}_i) \Rightarrow (\mathbf{F}\,\mathsf{D}_i))$$

The distributed algorithm $A^c_{of} = P_1 \otimes P_2$, where P_1 is the process algorithm described by Figure 1 and P_2 is the symmetric of P_1 obtained by replacing the action $\mathbf{re}(2)$ actions by $\mathbf{re}(1)$, is such that $A^c_{of} \models \Phi^c_{\text{agree}} \wedge \Phi^c_{\text{valid}} \wedge \Phi_{\text{of}}$.

Finally, another interesting context is the one corresponding to a round-robin scheduling policy. This context is given by the LTL formula, which basically states that if the n processes behave in a round-robin fashion, i.e. there are active one after another, then they all have to decide.

$$\Phi_{\text{rr}} \stackrel{\text{def}}{=} \Big[\mathbf{G}\Big(\bigwedge_{1 \leq i \leq n} (\mathsf{active}_i \Rightarrow \mathbf{X}\,\mathsf{active}_{(1+i\%n)})\Big)\Big] \Rightarrow \Big[\bigwedge_{1 \leq i \leq n} \Big(\mathbf{F}\,D_i\Big)\Big]$$

For the previously mentioned algorithm, we have $A^c_{of} \not\models \Phi_{\text{rr}}$, in fact as said in Section 1.2, if the processes are scheduled in a round-robin fashion and if their input values are different, then they will exchange their value forever and never decide. Note that we could easily define some Φ^k_{rr} formula to specify a round-robin policy where every process performs exactly k successive moves (instead of 1).

Specification for ε-agreement algorithms. We assume that the data set $\mathcal{D}$ is such that $\mathcal{D}_{\mathcal{I}}$ and $\mathcal{D}_{\mathcal{O}}$ are finite subset of $\mathbb{Q}$. We now present a variant of the ε-agreement. As for consensus, each process receives an initial value and the output values must respect the following criteria: (1) they should be between the smallest input value and the greatest one (*validity*) and (2) the outputs values all stand in an interval whose width is less or equal to ε (*agreemeent*). For instance, if we take $\mathcal{D}_{\mathcal{I}} = \{0, 1\}$ and $\mathcal{D}_{\mathcal{O}} = \{0, \frac{1}{2}, 1\}$, then if the two processes have input 0 and 1 respectively, the sets of accepted output values for $\frac{1}{2}$-agreement is $\{\{0\}, \{1\}, \{\frac{1}{2}\}, \{0, \frac{1}{2}\}, \{\frac{1}{2}, 1\}\}$. In this case, we can rewrite the formula for validity and agreement as follows:

$$\Phi^{\varepsilon}_{\text{valid}} \stackrel{\text{def}}{=} \mathbf{X}\Big[\bigvee_{d_m \leq d_M \in \mathcal{D}_{\mathcal{I}}} \Big[(\bigvee_{1 \leq i \leq n} \mathsf{In}^{d_m}_i) \wedge (\bigvee_{1 \leq i \leq n} \mathsf{In}^{d_M}_i) \wedge \mathbf{G}\Big[\Big(\bigwedge_{d < d_m \in \mathcal{D}_{\mathcal{O}}} \bigwedge_{1 \leq i \leq n} \neg\,\mathsf{Out}^d_i\Big) \wedge \Big(\bigwedge_{d > d_M \in \mathcal{D}_{\mathcal{O}}} \bigwedge_{1 \leq i \leq n} \neg\,\mathsf{Out}^d_i\Big)\Big]\Big]\Big]$$

And:

$$\Phi^{\varepsilon}_{\text{agree}} \stackrel{\text{def}}{=} \mathbf{G} \bigwedge_{1 \leq i \neq j \leq n} \Big((\mathsf{D}_i \wedge \mathsf{D}_j) \Rightarrow (\bigvee_{d, d' \in \mathcal{D}_{\mathcal{O}} s.t. |d' - d| \leq \varepsilon} \mathsf{Out}^d_i \wedge \mathsf{Out}^{d'}_j)\Big)$$

For what concerns the specification of the execution context, we can take the same formulae Φ_{wf}, Φ_{of} and Φ_{rr} introduced previously for the consensus.

3 Synthesis

3.1 Problem

We wish to provide a methodology to synthesize automatically a distributed algorithm satisfying a specification given by a LTL formula. In this matter, we fix the number of processes n, the considered data set (which contains input and output values) $\mathcal{D}$ and the set of memory values M for each process. A process algorithm P is said to use memory M iff $P = (M, \delta)$. A distributed algoritm $A = P_1 \otimes ... \otimes P_n$ uses memory M if for $i \in [1, n]$, the process P_i uses memory M. The **synthesis problem** can then be stated as follows:

Inputs: A number n of processes, a data set $\mathcal{D}$, a set of memory values M and a LTL formula Φ over $\mathsf{AP}^n_{\mathcal{D}}$

Output: Is there a n processes distributed algorithm A over $\mathcal{D}$ which uses memory M and such that $A \models \Phi$?

We propose a method to solve this decidability problem and in case of positive answer we are able to generate as well the corresponding distributed algorithm.

3.2 A set of universal Kripke structures for the synthesis problem

We show here how the synthesis problem boils down to find a specific Kripke structure which satisfies a specific LTL formula. In the sequel, we fix the parameters of our synthesis problem: a number n of processes, a data set $\mathcal{D}$, a set of memory values M and a LTL formula Φ_{spec} over $\mathsf{AP}^n_{\mathcal{D}}$. We build a Kripke structure $\mathcal{K}_{n,\mathcal{D},M}$ similar to the Kripke structure $\mathcal{K}_A$ associated to a distributed algorithm A but where the transition relation allows all the possible behaviors (all the possible move for every process in any configuration).

First, note that each process algorithm P for an environment of n processes over the data set $\mathcal{D}$ which uses memory M has the same set of process states $\mathcal{S}_P$. We denote $\mathcal{S} = \mathcal{D}_{\mathcal{I}} \cup (\mathcal{D}^n \times M)$ this set. Similarly each n processes distributed algorithm A over $\mathcal{D}$ which uses memory M has the same set of configurations $\mathcal{C}_A$ that we will denote simply $\mathcal{C}$. We recall that these configurations are of the form $(\mathbf{S}, \mathbf{R})$ with $S \in \mathcal{S}^n$ is a vector of n processes states and $\mathbf{R} \in \mathcal{D}^n$.

The Kripke structure $\mathcal{K}_{n,\mathcal{D},M}$ uses the set of atomic propositions $\mathsf{AP}^n_{\mathcal{D}} \cup \mathsf{AP}_{\mathcal{C},\mathcal{O}}$ where $\mathsf{AP}_{\mathcal{C},\mathcal{O}} = \{\mathsf{P}_{C,\mathbf{O}} \mid C \in \mathcal{C}, \mathbf{O} \in (\mathcal{D}_{\mathcal{O}} \cup \{\bot\})^n\}$ contains one atomic proposition for every pair made by a configuration C and vector of output values $\mathbf{O}$. Its states will be the same as $\mathcal{K}_A$ but for every possible actions there will be an outgoing edge. Formally, we have $\mathcal{K}_{n,\mathcal{D},M} = (Q, E, \ell, q_{\mathrm{init}})$ with:

- $Q = \{q_{\mathrm{init}}\} \cup (\mathcal{C} \times [0, n] \times (\mathcal{D}_{\mathcal{O}} \cup \{\bot\})^n)$ (as for $\mathcal{K}_A$)
- E is such that:

- $\big(q_{\text{init}}, ((\mathbf{In}, \bot), 0, \bot)\big) \in E$ for all initial configurations $(\mathbf{In}, \bot)$ in $\mathcal{D}^n_{\mathcal{I}} \times \{\bot\}^n$), (as for $\mathcal{K}_A$),
- $\big(((\mathbf{S}, \mathbf{R}), i, \mathbf{O}), ((\mathbf{S}', \mathbf{R}'), j, \mathbf{O}')\big) \in E$ iff $(\mathbf{S}, \mathbf{R}) \xRightarrow{j,(a,m)} (\mathbf{S}', \mathbf{R}')$ for some $(a, m) \in Act(\mathcal{D}, n) \times M$. And:
  - ∗ if $a = \mathbf{dec}(o)$ then $\mathbf{S}[j] = (\mathbf{V}, m)$ for $\mathbf{V} \in \mathcal{D}^n$ and $\mathbf{O}'[j] = o$ and $\mathbf{O}'[k] = \mathbf{O}[k]$ for all $k \in [1, n] \setminus \{j\}$, otherwise $\mathbf{O} = \mathbf{O}'$ (the memory cells does not change once the decision is fixed),
  - ∗ if $\mathbf{O}[j] \neq \bot$, then $a = \mathbf{dec}(\mathbf{O}[j])$ (the decision cannot change, no other action can be performed).

– the labelling function ℓ is defined the same way as in $\mathcal{K}_A$ for the atomic propositions in $\mathsf{AP}^n_{\mathcal{D}}$ and $\mathsf{P}_{C,O} \in \ell((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ iff $C = (\mathbf{S}, \mathbf{R})$ and $O = \mathbf{O}$.

Hence the relation E simulates all the possible moves from any configuration $(\mathbf{S}, \mathbf{R})$ and the Kripke structure $\mathcal{K}_{n,\mathcal{D},M}$ contains all possible executions of any n processes algorithms over $\mathcal{D}$ using memory M.

Defining an algorithm consists in selecting exactly one action for each process in every configuration. Here we do this by adding to the structure extra atomic propositions $\mathsf{P}^i_{(a,m)}$ with $1 \leq i \leq n$ and $(a, m) \in Act(\mathcal{D}, n) \times M$ which specifies for each configuration what should be the next move of process i. We denote by $\mathsf{AP}^n_{Act,M}$ this set of new atomic propositions. An *algorithm labelling* for $\mathcal{K}_{n,\mathcal{D},M}$ is then simply a function $\ell' : Q \mapsto 2^{\mathsf{AP}^n_{Act,M}}$. We denote by $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ the Kripke structure obtained by adding to $\mathcal{K}_{n,\mathcal{D},M}$ the extra labelling provided by ℓ'. When defining such an algorithm labelling, we need to be careful that it corresponds effectively to a distributed algorithm: our processes are deterministic (only one action is allowed for P_i in some configuration) and a process has to choose the same action when its local view is identical. Such an algorithm labelling ℓ' is said to be *consistent* iff the following conditions are respected:

1. $\ell'(q_{\text{init}}) = \emptyset$,
2. for all $((\mathbf{S}, \mathbf{R}), i, \mathbf{O}) \in Q$, for all $j \in [1, n]$ there exists a unique $\mathsf{P}^j_{(a,m)} \in \ell'((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$, each process has exactly one move in each configuration,
3. for all $((\mathbf{S}, \mathbf{R}), i, \mathbf{O}), ((\mathbf{S}', \mathbf{R}'), j, \mathbf{O}') \in Q$, if $\mathbf{S}[k] = \mathbf{S}'[k]$ and if $\mathsf{P}^k_{(a,m)} \in \ell'((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ then $\mathsf{P}^k_{(a,m)} \in \ell'((\mathbf{S}', \mathbf{R})', j, \mathbf{O}')$, i.e. in all configuration with the same state of process k, the moves of process k must be the same.

A consistent algorithm labelling ℓ' induces then a distributed algorithm $A^{\ell'} = P_1 \otimes \ldots \otimes P_n$ where for all $j \in [1, n]$, we have $P_i = (M, \delta_i)$ and $\delta_i(s) = (a, m)$ iff for all configurations $((\mathbf{S}, \mathbf{R}), j, \mathbf{O}) \in Q$ such that $\mathbf{S}[i] = s$, we have $\mathsf{P}^i_{(a,m)} \in \ell'(((\mathbf{S}, \mathbf{R}), j, \mathbf{O}))$. Conditions 1. to 3. ensure that this definition is well-founded.

To check by the analysis of the Kripke structure $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ whether the algorithm $A^{\ell'}$ induced by a consistent algorithm labelling satisfies the specification Φ_{spec}, we have to find a way to extract from $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ the execution corresponding to $A^{\ell'}$. This can be achieved by the following LTL formula:

$$\Phi_{\text{out}} \stackrel{\text{def}}{=} \mathbf{X}\,\mathbf{G}\Big[\bigvee_{C \in \mathcal{C}} \bigvee_{\mathbf{O} \in \mathcal{O}} \bigvee_{i,a,m} \Big(\mathsf{P}^i_{(a,m)} \wedge \mathsf{P}_{C,\mathbf{O}} \wedge \mathbf{X}(\mathsf{active}_i \Rightarrow \mathsf{P}_{\text{Next}(C,\mathbf{O},i,a,m)})\Big)\Big]$$

where $\text{Next}(C, \mathbf{O}, i, a, m)$ is the (unique) extended configuration $(C', \mathbf{O}')$ such that $C \xrightarrow{i,(a,m)} C'$ and $\mathbf{O}[j] = \mathbf{O}'[j]$ for all $j \neq i$ and $\mathbf{O}'[i] = o$ if $a = \mathbf{dec}(o)$ otherwise $\mathbf{O}'[i] = \perp$. We can then combine Φ_{out} with the correctness specification Φ_{spec} to check in $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ whether the executions of $A^{\ell'}$ (which are the executions of $\mathcal{K}_{A^{\ell'}}$) satisfy Φ_{spec}.

Proposition 2. *Given a consistent algorithm labelling ℓ' and its induced distributed algorithm $A^{\ell'}$,*

$$A^{\ell'} \models \Phi_{spec} \quad \textit{iff} \quad \mathcal{K}^{\ell'}_{n,\mathcal{D},M} \models \Phi_{out} \Rightarrow \Phi_{spec}$$

Sketch of proof. To prove this it is enough to see that the control states of $\mathcal{K}_{A^{\ell'}}$ and of $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ are the same and that any infinite sequence of such states ρ beginning in q_{init} is an execution in $\mathcal{K}_{A^{\ell'}}$ iff it is an execution in $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ verifying Φ_{out}. $\square$

Consequently, to solve the synthesis problem it is enough to find a consistent algorithm labelling ℓ' such that $\mathcal{K}^{\ell'}_{n,\mathcal{D},M} \models \Phi_{\text{out}} \Rightarrow \Phi_{\text{spec}}$. Note that as explained before this produces exactly the correct algorithm $A^{\ell'}$. We have hence a decision procedure for the synthesis problem: it reduces to some instances of model-checking problem for LTL formulae.

4 Experiments

We have implemented a prototype to automatically synthesize algorithms for consensus and ε-agreement problems. For this we use the SMT solver Z3[4]: it is now classical to use SAT solver for model-checking [1] and it was natural to consider this approach especially because we need to add an existential quantification over the atomic propositions encoding the moves of the processes [2]. Our prototype is however a bit different from the theoretical framework explained in Section 3 and we explain here the main ideas behing its implementation.

First, the implementation does not consider general (quantified) LTL formulas but encodes directly the considered problem (consensus or ε-agreement) for a set of parameters provided by the user into a Z3-program, and the result provided by the SMT solver Z3 is then *automatically* analysed in order to get algorithms for processes.

We now sketch the main aspects of the reduction to Z3. The code starts by existentially quantifying over the action functions for each process: an action function δ^p for a process p is encoded as an integer value δ^p_s for every process state s which gives the next action to performed. In Z3, such a δ^p_s is a bitvector (whose size is $\log_2(|Act(\mathcal{D}, n) \times M| + 1)$). It remains to encode the different properties we want to ensure (depending on the considered problem). Here are several examples:

[2] We do not describe here the reduction: it uses standard techniques for encoding LTL formulae to SAT instance.

– To deal with the formula Φ^c_{agree} for the consensus, we use a set of Boolean constants (one for every global configuration C). Their truth value can be easily defined as true when all processes in C have terminated and decided the same value, or as false when at least two processes have decided different values in C. For the other cases, we add constraints stating that the value associated with C equals true when for every successor (here a successor is any configuration reachable after an action (a, m) of some process p such that this action (a, m) corresponds to the value δ^p_s where s is the state of p in C). It remains to add a last constraint: for every initial configuration C_0, the constant associated with Φ^c_{agree} has to be true. Note that this definition is based on the fact that the property is an invariant: we want to ensure that no reachable configuration violates a local property.
– Encoding the formula Φ^c_{valid} follows the same approach: we use a boolean value for every configuration C and for every input data d, and define their truth value in such a way that it is true iff the value d cannot be decided in the next configurations. If some process has already decided d in C, the constant equals to false. If all processes have decided and no one choose d, it is true. Otherwise a positive value requires that for every successor C', the constants are also true. Finally we add constraints specifying for every initial configuration C_0 the values d that cannot be chosen by requiring that their corresponding values are true.
– The obstruction free context Φ_{of} is encoded as follows: we need two sets of constants for every process p. The first set contains one integer value (encoded as a bitvector in Z3) for every configuration and it is defined in order to be the number of moves that process p has to perform (alone) to decide (and terminate). This distance is bounded by the number of states nb_{state} of process p (and we use the value nb_{state} to represent the non-termination of the process). In addition, we consider a set of boolean values (one for every configuration) which are defined in order to equal to true iff for every reachable configuration from C, the computed distance is strictly less than nb_{loc}.
– Encoding the wait-free context uses the same idea. We have to verify that from every reachable configuration, every process will terminate (for this we use the fact that when a process decides a value, it does not perform action anymore, and then other processes progress). Note that in this case, the bound on the distance is the number of *global* configurations.

In addition to this encoding, we can also use standard techniques of bounded model-checking by fixing a smaller bound for the computation of the distances described above. When this is done, the program may provide an algorithm, or answer that an algorithm **with this bound** does not exist (it remains to try with a greater bound). This heuristic is crucial to synthesize algorithms in many cases (the computation of distances is quite expensive since it is connected to the number of states or configurations).

The parameters of our prototype are then: (1) the number of processes: n, (2) the range of initial values and the range of possible values in registers, (3)

the size of the processess memory, (4) the types of scheduling policy (wait free, obstruction free, round-robin, or a combination of them), and (5) the value of ε for the ε-agreement problem. Finally one can ask for symmetric programs (each process has the same action function) and in the following we only consider symmetric solutions.

State explosion problem. As explained in previous sections, we are faced with a huge complexity. For example, with 2 processes, two possible initial values and a memory size equals to 2, there are more than 450 configurations for the distributed algorithms. If we consider 3 processes, 2 initial values et a memory size equals to 3, we get more than 240 *thousands* configurations ! This gap explains why our prototype only provides algorithms for 2 processes. Note that even for the case $n = 2$, the complete encoding of the problem may use several thousands of variables in the Z3 code, and the SMT solver succeeds in providing a result. Of course, the implementation of our prototype in its current form is quite naive and some efficiency improvements are possible.

Moreover note that our prototype is often more efficient for finding algorithms when they exist than for proving that no algorithm within the resource fixed by the parameters[3] exists. First it is often easier to find a valuation than verifying that no valuation exists, and secondly we can use heuristics to accelerate the procedure (for example by bounding the length of computations: in this case, if a valuation is found, we can stop, otherwise we have to try again with different settings). This fact can be seen as a variant of a well-known phenomenon in sat-based model-checking: it is usually very efficient to find a bug (that is an execution satisfying or not a formula), but it is not the case to prove full verification.

Consensus. For 2 components, 2 initial and final values, a memory of size 2 and the obstruction free policy, we get the algorithm of Section 1.2 (Figure 1) except that the processes use their register to write the value they *do not* plan to decide (it is clearly symmetric to the previous algorithm). Note that the size of memory is important: there is no algorithm with memory of size 1: indeed we need to distinguish the configuration $(0, 0)$ (the proper register equals to 0 and the last read value of the register of other process is 0) when it is reached after a Read (both process agree on the value to decide) and when it is reached after a Write(0) performed by the process to update its register in order to agree with the other process. This absence of algorithm with a memory of size 1 corresponds to an UNSAT result for the program: the formula Φ_{synth} with these parameters is not satisfiable. When we tried to look for algorithms for wait-free case, we found no solution with our program: indeed we know that there is no such algorithms !

More interestingly we can ask for a program correct w.r.t. several execution contexts. For example, we can ask for program correct w.r.t. obstruction free, round-robin for one step and also round-robin for two steps. The program generates [4] the algorithm depicted in Figure 2 (we follow the same presentation

[3] Note that we cannot prove that no algorithm exists, but only that no algorithm *with this memory bound* exists if the corresponding SAT instance has no solution.

[4] It takes few seconds to produce the algorithm on a standard laptop.

as in Section 1 for the algorithm and since we have only two processes, we use $(\mathbf{re}, -)$ instead of $(\mathbf{re}(1), -)$: a read operation always deals with the other process).

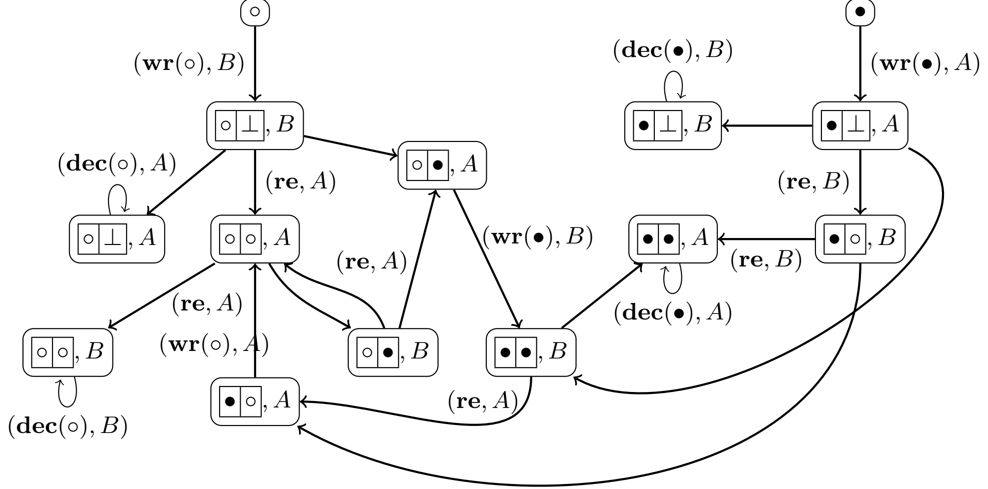

Fig. 2. View of a process algorithm P for consensus, w.r.t. to obstruction free and round-robin 1 and 2.

ε-agreement. For this problem, we have to fix ε. In Figure 3, we present an algorithm for $\frac{1}{5}$-agreement for 2 processes, with initial values $\{0, 1\}$ and memory 3. The set of possible decision values is $\{0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1\}$. Note that this algorithm works for the wait-free execution context, and therefore also for round-robin (for any step) and for obstruction free. Here the memory size equals to 3: this is illustrated by the fact that the configuration $(0, 0)$ (the register's value is 0 and the last read value from the other process is 0) appears in three nodes.

5 Conclusion

We have shown here that in theory it is possible to solve the synthesis problem for distributed algorithm as soon as we fix the set of data that can be written in the registers and the memory needed by each process in the algorithm. However even if this problem is decidable, our method has to face two different problems: first, it does not scale and second, when the answer to the synthesis problem is negative, we cannot conclude that there is no algorithm at all. In the future, we will study more intensively whether for some specific cases we can decide

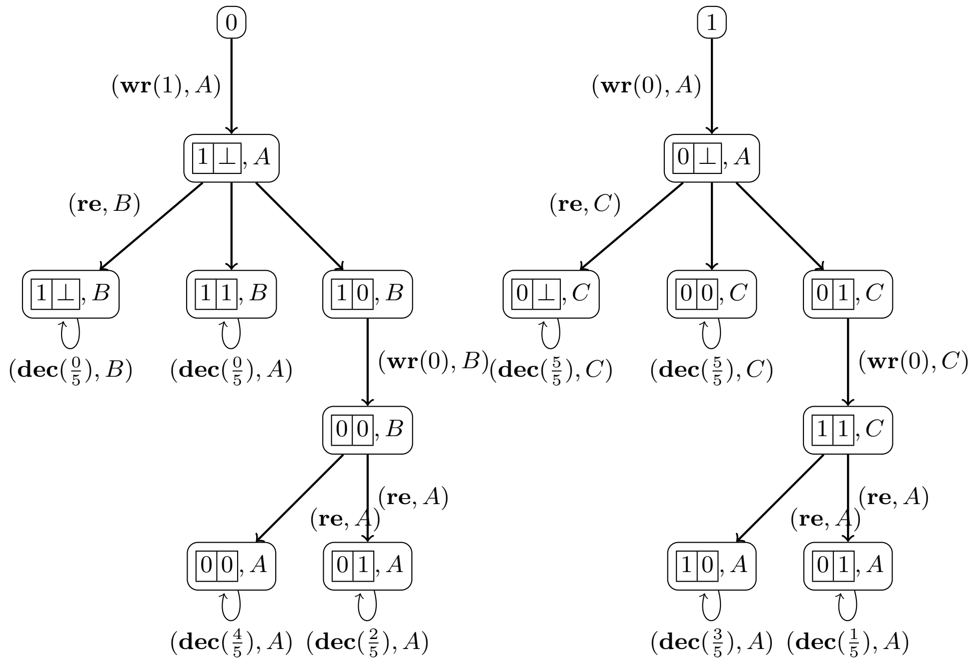

Fig. 3. View of a process algorithm P for $\frac{1}{5}$-agreement, w.r.t. wait free scheduling.

the existence of a distributed algorithm satisfying a given specification without fixing any restrictions on the exchanged data or on the size of the algorithms. We believe that for some specific distributed problems, this is in fact feasible.

References

1. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
2. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. C. Kozen, editor, *LOP'81*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1982.
3. D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA + proofs. In *FM'12*, volume 7436 of *LNCS*, pages 147–154. Springer, 2012.
4. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
5. E. A. Emerson. Temporal and modal logic. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, 1990.
6. M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
7. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *LNCS*. Springer, 2002.
8. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
9. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS'03*, pages 522–529, 2003.
10. M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.
11. G. J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.
12. M. Lazic, I. Konnov, J. Widder, and R. Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS'17*, volume 95 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
13. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Comp. Soc. Press, Oct.-Nov. 1977.
14. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *SOP'82*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, Apr. 1982.
15. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS'86*, pages 332–344. IEEE Computer Society, 1986.