IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 25 May 2023, accepted 14 June 2023, date of publication 19 June 2023, date of current version 26 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3287405

RESEARCH ARTICLE

Synthesis of Fault-Tolerant Reliable Broadcast Algorithms With Reinforcement Learning

DIOGO VAZ, (Graduate Student Member, IEEE), DAVID R. MATOS, (Member, IEEE), MIGUEL L. PARDAL, (Member, IEEE), AND MIGUEL CORREIA, (Senior Member, IEEE)

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisbon, Portugal

Corresponding author: Diogo Vaz (diogo.vaz@tecnico.ulisboa.pt)

This work was supported by the National Funds through Fundação para a Ciência e a Tecnologia (FCT) under Grant UIDB/50021/2020 (INESC-ID) and Grant 2022.10788.BD.

ABSTRACT Fault-tolerant algorithms, such as *Reliable Broadcast*, assure the correct operation of modern distributed systems, even when some of the system nodes fail. However, the development of distributed algorithms is a manual and complex process, where slight changes in requirements can require a complete redesign of the algorithm. To automate the process of developing such algorithms, this work presents a new approach that uses Reinforcement Learning to synthesize correct and efficient fault-tolerant distributed algorithms. This work shows the first application of the approach on the synthesis of fault-tolerant Reliable Broadcast algorithms. The presented technique is capable of synthesizing correct and efficient algorithms with the same performance as others available in the literature as well as a new *Byzantine* tolerant algorithm, in only 12, 000 learning episodes. Based on the success of this implementation, we aim, in the future, to extend this technique to other distributed algorithms such as Consensus.

INDEX TERMS Fault-tolerant distributed algorithms, reliable broadcast, program synthesis, reinforcement learning, program verification, model-checking.

I. INTRODUCTION

Distributed systems are made of multiple components interconnected by communication networks. Examples of modern and widely used distributed systems are cloud applications [2] and blockchains [65]. During normal operation, some of these components may fail, e.g., due to power loss, software bugs, or malicious attacks. These faults can compromise the normal functioning of the entire system. Therefore, it is necessary to provide fault-tolerance properties so that the distributed system can maintain its normal execution, even in the presence of faults. For this purpose, it is important to design and implement *fault-tolerant distributed algorithms* [11], [61].

Fault-tolerant distributed algorithms have been widely studied and developed over the years [6], [8], [9], [16], [47], exploring different aspects such as the problem to solve [6], [47], the fault model [26] or the system architecture [8], [16]. However, the process of studying and designing a fault-tolerant algorithm is a *manual*, *time-consuming* and *complex* endeavor [26]. This is especially true when malicious faults are considered, where algorithms are complicated, and slight changes often require a complete redesign. Furthermore, current research on fault-tolerant algorithms does not focus only on *correctness*: *efficiency* is also a very important concern, leading to an increase in the complexity of the algorithm development process.

The journey to develop a fault-tolerant algorithm starts with defining the problem the algorithm will solve, e.g., Reliable Broadcast, shortened in this paper as RBcast. Next, assumptions about the environment (system model) are made, e.g., if the system is synchronous or asynchronous (i.e., if there are assumptions about time or not) and the failures that can occur. The researchers then consider the strategy to design the distributed algorithm. In this stage, in addition to the difficulty in creating the algorithm, researchers may be biased by previous related papers and algorithms. After a trial-and-error process, the distributed algorithm is obtained. This also involves a verification process to assess whether the

The associate editor coordinating the review of this manuscript and approving it for publication was Yunlong Cai.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

generated algorithm achieves the goal and solves the problem correctly. This can be done by writing a manual proof or by doing verification using a model checker [14] or a theorem prover [7].

In this work, we present a new approach to automate the process of synthesizing fault-tolerant Reliable Broadcast algorithms. Although only applied to this class of algorithms, we believe this approach can be extended to the synthesis of other distributed algorithms. Therefore, this can be the beginning of a new path of research in the field of distributed computing that can be adopted to improve and design new distributed systems. This comes at a time when there is an increasing need for higher scale and even more geographically dispersed deployments of distributed systems with the emergence of technologies such as *Internet of Things* (IoT) [42] and *Distributed Ledger technologies* (DLTs) [40].

Our technique is based on the idea of *Generative Artificial Intelligence (AI)*: from a specific description of the algorithm needed and some previous knowledge, the approach will be capable of synthesizing a correct, efficient, potentially new algorithm. Besides first appearing associated with *Generative Adversarial Networks* (GANs) [34], more recently, Generative AI evolved to a broader scope of AI solutions capable of generating new data based on pre-existing knowledge [25], [41], [45]. Solutions such as *ChatGPT*[1] and *GitHub CoPilot*[2] gained notoriety with the generation of text and code.

The core of our approach is an iterative process with two phases: *generation*, to obtain candidate algorithms, and *verification*, to evaluate their correctness. The process goes on discovering new algorithms and converging towards one that is correct and efficient in terms of a set of metrics. For the first phase, *generation* phase, we propose an agent that, based on a machine learning technique, *Reinforcement Learning* [51], [70], [73], synthetizes distributed algorithms. For the second phase, *verification* phase, we propose an additional agent that assesses the correctness of the generated algorithms. The search for a verification process for distributed algorithms has been ongoing for years [23], [24], [49], [66]. We identified a group of possible frameworks that the agent can use, such as the TLA+ language and tools [53], the Spin framework with the PROMELA language [46], or the ByMC framework [52], to new a few. We opted to use Spin/PROMELA, as explained later (see section VIII).

There are already a few works an automatic synthesis of distributed algorithms: three focused on mutual exclusion algorithms [4], [36], [37], two on consensus algorithms [30], [74], one on fault-tolerant distributed algorithms [56], and one on leader election algorithms [38]. However, none of them uses machine learning techniques. As far as we know, machine learning techniques have been used only for the generation of local, non-distributed code, mostly using supervised and unsupervised machine learning techniques [3], [18], [58], [68]. In this work, we use Reinforcement Learning that allows running an agent following a trial and error process, without the necessity of having a dataset with fault-tolerant distributed algorithms (i.e., it is not supervised). Such a dataset would be laborious and complex to create, as it would require a large set of pre-existing algorithms that solved the problem. In relation to Reinforcement Learning, the most interesting works that we have found were: a framework to improve pre-trained language models for program synthesis tasks through deep reinforcement learning [57], a work that uses reinforcement learning to generate matrix multiplication algorithms [22], a work that uses reinforcement learning to generate tests for Android GUI applications [39], and a work that uses reinforcement learning coupled with deductive reasoning for program synthesis [28]. However, all these works are very different from ours. To the best of our knowledge, this work is the first to present an approach based on a form of machine learning, Reinforcement Learning, in the field of distributed computing, to generate distributed algorithms.

The main contributions of the paper are: (1) a new approach for synthesize distributed fault-tolerant algorithms using machine learning, instead of manual development by human beings, (2) an intelligent agent to generate RBcast distributed algorithms, and (3) an experimental evaluation of the approach and the agent, showing a generation of correct RBcast algorithms and a new efficient Byzantine-tolerant algorithm.

II. RELIABLE BROADCAST

This section presents RBcast, the distributed problem for which we want to find algorithms to solve.

A. SYSTEM MODEL

The system model considered for the RBcast algorithms we want to synthesize is inspired by the modular approach to fault-tolerant problems by Hadzilacos and Toueg [26]. The system is composed of a static group of N processes, i.e., there are no joins or leaves during execution. We assume a *fully connected point-to-point* network, i.e., that all processes are connected with each other through *links* and communicate by passing messages. We also assume that the system is *asynchronous*, i.e., that the communication delays are neither upper-bounded nor respect a global stabilization time.

A *process* is the actor of the distributed system that executes a set of specific ordered *actions*, together designated as an *algorithm*. All processes in the system execute the same algorithm.

Communication links allow processes to exchange *messages* by transporting the message from sender to the receiver. We assume that the links are *reliable*, *authenticated*, and *provide integrity on the messages*, meaning that there are no corrupted, lost or duplicated messages. However, messages may arrive out of order.

Processes use *messages* to share data between them. Typically, a message contains data such as the message content

[1] https://openai.com/blog/chatgpt/
[2] https://github.com/features/copilot

(used by the logic of the system) and an identifier that can contain *protocol type*, *sender*, and *sequence number*.

System processes can be correct or faulty. We consider *three failure modes*: No-Failure, Crash-Failure, and Byzantine-Failure. In the simplest, *No-Failure*, we assume that there are no failures. In the *Crash-Failure mode* [5], [20], processes may stop operating and never recover. Assuming a system with $N \in \mathbb{N}$ processes, the maximum number of faulty processes $F \in \mathbb{N}$ due to a crash failure that can be tolerated in the system is $F = \lfloor (N-1)/2 \rfloor$ [10]. In the *Byzantine-Failure mode* [54] faulty processes may have arbitrary behavior, e.g., they may execute other actions not defined by the algorithm or even not execute any action at all. Unlike a crash failure, when a process suffers a Byzantine failure (or, "is Byzantine"), it can continue to work. Assuming a system with N processes, the maximum number of faulty processes F due to a Byzantine failure that can be tolerated in the system is $F = \lfloor (N-1)/3 \rfloor$ [10], [16].

B. PROBLEM DEFINITION

A RBcast algorithm ensures, essentially, that every message broadcast by a *correct* process (an *RB-Broadcast* event) is eventually delivered by all *correct* processes. The protocol is defined by the following properties[3] [16], [26]:

- *RB-Agreement*: if a correct process delivers a message m, then all correct processes will eventually deliver the same message m;
- *RB-Validity*: if a correct process broadcasts a message m, then it will eventually deliver that message m;
- *RB-Integrity*: for any message m, every correct process delivers m at most once and only if m was previously broadcast by some *(correct or incorrect)* process.

The term *correct* refers to a process that follows the algorithm. Otherwise, we call it incorrect or faulty.

Sometimes a system requires stronger properties than these, e.g., properties about the order of message delivery. To provide these extra properties, variants of the RBcast problem were specified, such as *FIFO Broadcast*, where messages broadcast by the same process are delivered in the order they were broadcast; *Atomic Broadcast*, that assures all processes deliver the same messages in the same order [16], [26], [50]; or *Terminating Reliable Broadcast*, that assures that all correct processes eventually deliver something [43]. Nevertheless, in this work, we use only the default RBcast to illustrate our approach to automated algorithm generation.

III. LEARNING THE RBcast ALGORITHM

Machine learning techniques can be divided into three broad classes: *supervised machine learning* – to learn from labeled data, *unsupervised machine learning* – to learn without labeled data – and *reinforcement learning* – to learn from a reward mechanism. In this task of learning how to solve a distributed problem, we decided to emulate the *trial and error* strategy used by human researchers to solve this problem. Therefore, we decided to use *Reinforcement Learning* [51], [70], [73].

[3]Other works may present an equivalent definition based on additional properties such as No-Duplication, Consistency, or Termination.

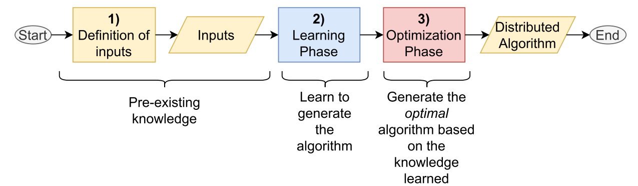

FIGURE 1. Process/dataflow of the proposed solution.

Reinforcement Learning is based on the idea of an *agent* choosing *actions* in specific *states*. The states are the observations that the agent receives from the *environment* where it acts. The way in which the agent acts is defined by the *policy*, in this case, a map from perceived states to actions to be taken in those states. Then, by choosing an action in a state, the agent will receive a *reward*, reflecting its choice. With time, the agent will start learning the *value* of each state, i.e., the total amount of reward the agent can expect to accumulate over the future, starting from that state. While rewards are *short-term* indicators, the values reflect the *long-term* desirability of that state, taking into account the states that are likely to follow and the rewards associated with them. We use *model-free* Reinforcement Learning, in the sense that the agent does not create a representation of the behavior of the environment, unlike model-based approaches. A model-based agent would need to have a model of the dynamics of the environment, allowing to predict state transitions and rewards, e.g., given the current algorithm, the agent could plan future actions and their rewards.

In this work, we propose a novel approach to synthesize fault-tolerant Reliable Broadcast algorithms based on the Reinforcement Learning technique. Figure 1 presents the entire process of the approach, divided into three phases:

1) *Definition of inputs:* in this phase, the user defines the inputs in order to describe the specifications of the algorithm to be synthesized. This is the phase where the user will input some existing knowledge of the area of distributed algorithms (e.g. the fault tolerance ratio);
2) *Learning Phase:* in this phase, the approach will learn to solve the problem by experience, through the generation of multiple algorithms, both correct or incorrect;
3) *Optimization Phase:* in this last phase, the proposed approach will generate a correct and efficient algorithm – the *optimal* algorithm – based on the knowledge obtained from the Learning Phase.

Although we believe and aim, in the future, to apply this approach to other distributed algorithms, in this work, we present the first study applied to a single problem: the Reliable Broadcast problem. In the following sections, we explain the specific synthesis process and the definitions of the algorithm adopted in this work, namely, the structure, messages, types, conditions, and efficiency.

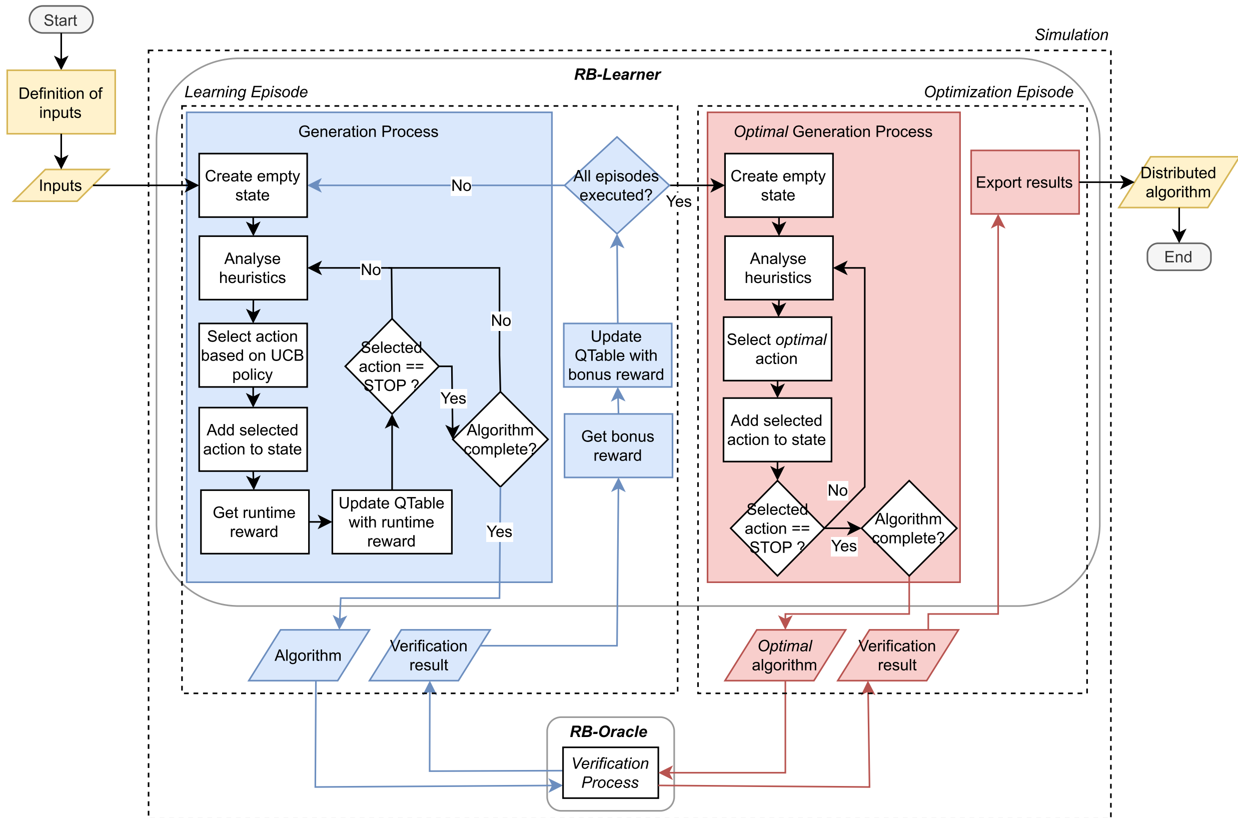

FIGURE 2. Process/dataflow of synthesizing an optimal algorithm. The *learning phase* is represented by the blue components and the *optimization phase* is represented by the red components.

TABLE 1. Inputs used in the paper for synthesis of RBcast algorithms.

Simulation inputs	Possible Values	Generation Process inputs	Possible Values	Verification Process inputs	Possible Values
Number of simulations	{1,2,3,...}	Rewards	{...,-1,0,1,..}	Failure modes to model	{No-Failure, Crash-Failure or Byzantine-Failure}
Number of learning episodes	{1,2,3,...}	Heuristics to be applied	(cf. Sec. IV-E)	System models to use	{e.g. [$N = 3, F = 0$], [$N = 3, F = 1$], [$N = 4, F = 1$]} for No-,Crash- and and Byzantine-Failure, respectively
				Properties to be verified	{RB-Validity, RB-Agreement, RB-Integrity}

A. SYNTHESIS PROCESS

Our solution considers an *agent* that has the goal of synthesizing correct and efficient RBcast algorithms, i.e., algorithms that satisfy the RBcast properties (cf. Section II-B) and minimizes the efficiency metrics (cf. Section III-E). The solution is composed of a main agent, *RB-Learner*, that collaborates with an auxiliary agent, *RB-Oracle*, as represented in Figure 2. The entire execution of the solution for one algorithm is designated as a *simulation*. The process starts with the definition of the *inputs* of the simulation. Table 1 summarizes these inputs. The *simulation inputs* include the number of simulations and learning episodes to run. The *generation process inputs* include the rewards and heuristics to be used, the last two containing domain knowledge about the problem (see Sections IV-C and IV-E). The *verification process inputs* include the specifications to verify the algorithm, such as the failures to model, the system architecture to model, and the properties to be verified.

The simulation starts with the *learning phase* which is composed of the execution of a set of *learning episodes*. In each of these episodes, the RB-Learner agent runs the *generation process*, i.e., generates one algorithm and gives the algorithm to the RB-Oracle agent, which executes the *verification process*, i.e., assesses whether the generated algorithm actually solves the problem. Then, the RB-Oracle returns the verification result to the RB-Learner, which it uses as part of the learning phase. When all learning episodes are executed, the agent runs one *optimization episode* composed by the execution of a single *optimal generation process*. In this episode, the agent generates the *optimal algorithm* based on the knowledge obtained from the learning episodes. The optimal algorithm may not be correct because the agent may

not be able to learn to solve the problem, so the verification process is executed one final time to analyze the correctness of the optimal algorithm. In the rest of the work, we often use the term *episodes* to refer to learning episodes, as the optimization episode is a single one and it has a specific goal.

With this technique, the idea is that the agent will be able to learn/synthesize a correct and efficient algorithm by generating multiple algorithms – either correct or incorrect – without the need for prior knowledge of the state of the art in RBcast.

B. STRUCTURE

RBcast has been studied extensively over the years [6], [8], [9], [16], [47], [48]. The papers present RBcast algorithms with different structures. For example, Bracha and Toueg presented algorithms organized in execution steps [9], [10], while more recent work favours a structure based on event handling routines [47], [48]. We follow the latter, an event-oriented structure.

We assume that the structure of the algorithm is composed of two events: the *RB-Broadcast* event, triggered only once by the process that starts the execution of the algorithm; and a *receive* event, triggered every time a process receives a message. Each event can contain a set of *actions*, i.e., instructions that are executed when the event is triggered. The execution of the algorithm ends when there are no more messages to be received.

C. MESSAGES AND TYPES

As discussed in Section II-A, messages contain multiple elements, e.g., content, sender, and protocol type. To differentiate between the different communication steps of the algorithm, the messages must have a specific element called *type*. In this work, we follow previous RBcast works [9], [48] by representing messages in the format < *t,m*>, where *t* symbolizes the type of the message, and *m* the message itself. Previous works define types as words like *echo* or *init* [9], [47], but in this work, we designate types as *type0*, *type1*, *type2*, etc. The reason for this option is that the types are generated automatically, not by a human. But afterward, we can translate the *type0* into *init* and the *type1* into *echo*.

D. INSTRUCTIONS AND CONDITIONS

Instructions are statements defining behavior that can be executed by the algorithm. Conditions are statements used to evaluate when a specific instruction can be executed and are associated with the *if* clause. To solve Reliable Broadcast, we have defined instructions related to *sending* and *delivering* messages and defined 2 types of conditions: the *true* and the *threshold* conditions. The true condition is a neutral condition representing an always true behavior. Threshold conditions are defined by two properties: the *message* < *t,m*> and the *threshold* expression, which is the number of messages needed to satisfy the condition. For example, the statement

TABLE 2. Actions available to the RB-Learner.

Instruction	Message type	Condition	Message type
`SEND to all`(<t,m>)	*type0*, *type1*	*true*, 1, $F+1$, $(N+F)/2$, $N-F$	*type0*, *type1*
`SEND to neighbours`(<t,m>)	*type0*, *type1*	*true*, 1, $F+1$, $(N+F)/2$, $N-F$	*type0*, *type1*
`SEND to myself`(<t,m>)	*type0*, *type1*	*true*, 1, $F+1$, $(N+F)/2$, $N-F$	*type0*, *type1*
`DELIVER`(m)	-	*true*, 1, $F+1$, $(N+F)/2$, $N-F$	*type0*, *type1*
`STOP`	-	*true*	-

`SEND to all` (< type1,m>) *if received* (< type0,m>) *from* $F+1$ *distinct processes* means that the instruction `SEND to all` (< type1,m>) can only be executed, i.e. send a message *m* of type *type1* to all the processes in the system, if the process has received a message *m* of *type0* from at least $F+1$ processes (the threshold is $F+1$). Each condition can be associated with different instructions: if the algorithm contains different instructions with the same condition and the condition is satisfied, all of them are executed.

E. EFFICIENCY

To analyze the efficiency of an algorithm, we adopt a model based on previous works [11], [67], where the efficiency is related to three properties: (1) the number of messages sent by the algorithm; (2) the number of communication steps (which in the case of RBcast is the number of types, as there are no loops) and; (3) the number of messages that have to be received for the algorithm to stop. All these metrics indirectly express the cost of computational power, storage, and network to execute the algorithm. As the algorithm sends more messages or contains more communication steps, then the process will need more storage for messages, spend more network resources, and take more time to execute the algorithm.

IV. RB-LEARNER

The RB-Learner agent uses Reinforcement Learning to learn not only an algorithm that solves the problem but also an algorithm that is efficient. Next, we explain the elements behind the learning process of the RB-Learner.

A. ACTIONS

An algorithm is composed of a set of event handling routines – two in the case of RBcast, each of which contains a sequence of actions. The RB-Learner selects an *action* from the set of possible actions to add to one of the routines. Each action has two components: *instruction* – the part of the action that is executed – and *condition* – a statement that must be true in order for the instruction to be executed. For example, in action

`SEND to all`(< type1,m>) if received (< type0,m>) from 1 distinct process

the instruction component is the left-hand part (in red) and the condition is the right-hand part (in blue, starting with the word "if"). Next, we present the instruction and condition components that we assume in the paper.

1) INSTRUCTION

We selected the following instruction components taken from previous work that solve the RBcast problem [9], [26], [47]:

- `SEND to all(`< t,m>`)`: sends the message < *t,m*> to all processes of the system (including itself);
- `SEND to neighbours(`< t,m>`)`: sends the message < *t,m*> to all processes of the system (excluding itself);
- `SEND to myself(`< t,m>`)`: sends the message < *t,m*> only to itself;
- `DELIVER`(m): delivers the message *m*;
- `STOP`: end the execution of the event handler.

In addition to these, we could think of a generic component to send a message to $N - X$ processes, for any $X > 0$ and $X \in \mathbb{N}$. However, this generality is found only in probabilistic algorithms [11], [21], which are out of the scope of this work.

2) CONDITIONS

In this work, each action contains a specific *condition* that defines when the instruction in the action is executed. The true condition is only defined by the word *true*, while each threshold condition is associated with a specific *threshold* and a *message type*. From a range of works analyzed [9], [26], [47], we selected four thresholds: waiting for 1, $F + 1$, $(N + F)/2$, and $N - F$ messages from different processes. We assume that two threshold conditions are equal if they wait for the same message type and have the same threshold.

3) MESSAGE TYPE

As explained above, messages contain a type: *type0*, *type1*, and so forth. One parameter of the problem of generating an algorithm is how many types of messages it uses. Clearly, the minimum number of types found in algorithms in the literature corresponds to the maximum number of types needed to solve the problem. For RBcast we found that the number is of two types [47]. Both `SEND` actions and conditions are influenced by a certain type. On the contrary, the *true* condition is not associated with any threshold or message type.

Table 2 presents all actions available to our agent, resulting in a total of $T = 64$ possible actions – the total number of possible combinations using the actions of the table. All actions are associated with all possible conditions except `STOP` that does not depend on messages being received (or, equivalently, it depends only on the *true* condition).

We also assume that each action contains an implicit condition that forbids the action to execute more than once for the same message. This means that if a process delivers a message with content $m = 1$, it will never deliver the same message again. The same applies to `SEND` actions. This inner condition is introduced by previous articles that present RBcast algorithms: for example, in [47] we have the conditions *not yet broadcast* or *not yet RB_delivered* and in [26] we have the condition *if p has not previously executed deliver(R,m)*.

B. STATES

A typical Reinforcement Learning agent interacts with an external *environment*. In our case, the environment is not external, but internal memory. This memory stores the actions already selected by the agent to form the algorithm. Specifically, a state is the sequence of actions selected by the agent up to that moment. By following this representation, the agent will be able to learn to select the best actions based on the ones that the algorithm already contains. Each state follows the algorithm structure defined in Section III-B, being composed of two event handlers and expressed as *State([])*. We assume that *State* A and *State* B are equal if *both contain the same actions, in the same number, in the same order and in the same event handlers*.

C. REWARDS

Rewards are used by the agent to learn which actions are suitable or not in each state – a technique called reward shaping [55]. In this work, the rewards are related to the *efficiency* (cf. Section III-E) and *correctness* (cf. Section II-B) of the algorithm: the most efficient correct algorithm will generate the best reward. Rewards are defined as part of the input, but it is important to mention that the agent synthesizes the most efficient algorithm not because of the absolute reward values that we have defined, but because of the relative values between them (the absolute rewards are the result of testing multiple possibilities).

The RB-Agent receives a reward in two moments: (1) every time the agent selects an action – *runtime reward* – and (2) when the agent receives the verification result from the RB-Oracle – *bonus reward*.

1) RUNTIME REWARDS

Runtime rewards are related to the *efficiency* of the produced algorithm, i.e., more efficient algorithms will generate the best rewards. Table 3 summarizes the values we empirically established for calculating these rewards. The `SEND` actions and the `DELIVER` action have a negative reward, as processes need to spend time and resources to execute these actions, so there is a cost involved. For the `SEND` actions, the `SEND to myself` action has a better reward than the `SEND to neighbours` and the `SEND to all` actions. This happens due to the number of sent messages metric: `SEND to myself` only sends 1 message, whereas the others involve sending $N - 1$ and N messages, respectively. The threshold of the conditions also influences the reward. Table 3 shows the rewards associated with each selected threshold (c.f Section IV-A2). Considering that $1 \leq F + 1 \leq (N + F)/2 \leq N - F$, the higher the threshold, the higher its cost since processes need to wait for more messages to execute the instruction of the action, where N is the total number of processes and F the maximum number of faulty processes. The *true* condition has a reward of 0 since does not involve any cost such as the need for a specific number of messages or message types.

TABLE 3. Rewards given to each instruction and each condition.

Instruction	Reward	Condition	Reward
SEND to myself(<t,m>)	-1	*true*	0
SEND to neighbours(<t,m>)	-2	1	-1
SEND to all(<t,m>)	-3	$F+1$	-2
DELIVER(<m>)	-1	$(N+F)/2$	-3
STOP	0	$N-F$	-4

Beyond what is presented in Table 3, the addition of a new message type to the algorithm also involves a (negative) reward. Specifically, the reward is added when a SEND action introduces a new type. Each new type has an increasing cost: *type0* is associated with the reward 0, *type1* is associated with reward -1, etc. The objective is for the agent to add the minimum number of new types to the algorithm, as each new type involves more communication.

The last aspect that influences the reward obtained by the agent is the event handler where the action is selected. We defined that each action selected for the *RB-Broadcast* event handler has an additional reward of 0, while the actions selected for the *receive* event handler have an additional reward of -1. These rewards favour the addition of actions to the *RB-Broadcast* event handler instead of the *receive* event handler; this bias is needed because *RB-Broadcast* is executed only once per execution of the algorithm, whereas *receive* is executed N times (one per process), meaning that an action on the *receive* event handler will have a greater impact on the efficiency of the algorithm when compared to an action on the *RB-Broadcast* event handler, e.g, a SEND to myself action will have a cost of 1 message if executed on the *RB-Broadcast* event handler, but a cost of $N * 1$ if executed on the *receive* event handler.

To summarize, consider the example where the agent chooses the action:

SEND to all($<$ type0,m$>$) if received ($<$ type0,m$>$) from $N-F$ distinct parties

The reward for this action will be: -3 (the SEND to all instruction) + 0 (*type0* sent) -4 (the $N-F$ threshold) = -7. Then, if the action is selected for the *RB-Broadcast* event handler, it will receive an additional reward of 0 (still a total of -7), while if selected for the *receive* event handler, it will receive an additional reward of -1 (total of -8).

2) BONUS REWARDS

After the algorithm is verified by the RB-Oracle, the RB-Learner receives the verification result from the RB-Oracle. The RB-Learner will use that result – correct or incorrect – to get a bonus reward or not. In case the algorithm is correct, there is a bonus of 100, from where we discount the *runtime rewards* accumulated during the generation. For example, if the agent generates a correct algorithm with a runtime reward accumulated of -14 during the state transitions of the generation process, the bonus will be $100 + (-14) = 86$. This allows the agent to receive a better bonus for the most efficient algorithms. In the case of an incorrect algorithm, the reward received by the agent will be -1: from the total number of possible algorithms, the number of incorrect algorithms will tend to be greater than the correct ones, so we do not want the agent to be severely penalized by finding an incorrect algorithm since some actions of an incorrect algorithm can still lead to a correct algorithm.

D. LEARNING AND OPTIMIZATION PHASES

This section explains one learning episode, from the Learning Phase, and its generation process that builds one algorithm and the optimization episode, from the Optimization Phase, and its optimal generation process that generates the optimal algorithm, both detailed in Figure 2.

During a learning episode, the *generation process* is executed. This process is composed of two development steps: the step of the *RB-Broadcast* event handler and the step of the *receive* event handler. Both steps are based on *Q-Learning* [72], a broadly adopted Reinforcement Learning algorithm. This algorithm uses a table designated *QTable* to map the values of each action to each state. Next, we explain how the development steps generate the entire algorithm during the generation process.

The generation process begins with the development phase of the RB-broadcast event handler. The agent starts with the internal state empty, i.e. an empty algorithm with *State([])*. Then, comes a loop based on the current internal state, where the agent analyses a set of heuristics (a topic we defer for Section IV-E) to discard the actions not suitable for the current state, from the set of all possible actions. Then, the agent selects one of the suitable actions based on a policy. In this work, the agent follows the *Upper Confidence Bound* (UCB) policy [70], a policy based on the idea of being *optimistic under uncertainty*. This policy works by defining a confidence boundary assigned to each action that decreases when the action is more frequently chosen, helping to solve the exploration/exploitation problem [15] – the dilemma of selecting new actions or keeping the same selection of actions as before. Then, the selected action is added to the current internal state and to the current event handler – in this case the *RB-Broadcast* – originating a new state. For example, if the agent is in state *State([])* and chooses action A, the action is added to the algorithm and the *RB-Broadcast* event handler, originating the new state *State([action A])*. Moreover, based on the action selected, the agent receives a runtime reward that it uses to update its learning base, the QTable, by associating the reward received with the action selected in the current state. Finally, the agent defines its current state as the new state and re-executes the analysis of the heuristics to get the suitable actions for this new state.

The agent continues to re-execute the development of the *RB-Broadcast* event handler until the moment when it chooses the STOP action. By choosing this action, the development step of the current event handler – in this case, the *RB-Broadcast* event handler – is completed, and the development step of the next event handler – the *receive* event handler

– is started. The agent executes this second development step but now adds the actions to the *receive* event handler, until the moment when it chooses the STOP action again. This marks the end of the development step of the *receive* event handler. Only after the two development steps – the development of the *RB-Broadcast* and the *receive* events – the algorithms is considered completed.

With the development of the algorithm completed, the generation process ends, and the RB-Learner gives the algorithm to the RB-Oracle, which in turn, will verify it. After verifying the algorithm, the RB-Oracle returns the verification result to the RB-Learner which receives a bonus based on whether the algorithm is correct or not. This bonus reward will also be used to update the QTable of the agent.

In the first learning episodes, the generation process will produce random algorithms, led by the policy that allows it to *explore* new actions and states. As the learning phase progresses, based on the values of the QTable, the policy used by the agent will lead it to *exploit* the actions that have the best value in each state, allowing it to converge to the most efficient algorithms. We use the terms explore and exploit with the precise meanings they have in Reinforcement Learning: explore is related to the search for new and unfamiliar states, while exploit refers to the examination of familiar states [15].

The agent performs the *optimal generation process episode* when all the learning episodes are executed. In this final episode, the agent generates the *optimal algorithm*. The process to generate this algorithm is identical to the generation process. However, instead of being guided by a policy, in the optimal generation process, the agent will always select the *optimal* action allowed in each state, based on the knowledge obtained during the learning episodes. To be more specific, the agent will generate the optimal algorithm by always selecting the action that gives the best reward, following a policy usually called the *greedy policy*. With the optimal algorithm generated, the RB-Learner gives it to the RB-Oracle to execute the verification process and analyze either if the optimal algorithm is correct or incorrect. After the verification process, the RB-Learner outputs the results of the simulation, including the distributed algorithm found and its correctness.

E. HEURISTICS

The number of possible algorithms grows exponentially with the base given by the maximum number of allowed actions T in the algorithm (a constant), i.e., has complexity $O(T^i)$, where i is the total number of possible actions. Although the agent has i actions to choose from, there are clearly some bad choices in some cases. For example, it is a bad option to choose STOP as the first action since that would generate an empty algorithm.

To reduce the explosion of possibilities and guide the agent during the generation process, we define a set of *heuristics* [59], [64] for the agent to avoid bad choices. Notice that the heuristics do not help in obtaining correct and efficient algorithms; they only reduce the number of possibilities to explore by discarding invalid actions in specific states and, consequently, reducing the time required to find solutions.

TABLE 4. Heuristics used on the generation process.

Heuristic	Definition
GH1	Do not allow repeated actions on the algorithm
GH2	Allow to define the actions available in each event handler
GH3	Allow to define the conditions available in each event handler
GH4	We can only use a SEND action for each type of message sent and condition
GH5	Messages sent on the *RB-Broadcast* event handler must be of type *type0*
GH6	Allow to define the minimum and maximum number of actions that each event handler can have
GH7	Only select an action that waits for a message type already sent on the algorithm
GH8	The algorithms generated must contain, at least one DELIVER action
GH9	The incorrect states are blocked, in order to never explore them again
GH10	Allows defining the maximum number of types that the algorithm can contain

Table 4 presents the generation heuristics (GH) that we use to guide the agent in the case of RBcast. These heuristics were defined on the basis of a logic of discarding undesirable actions. Every time the agent is in a state, the agent uses the heuristics to know which actions are available in this specific state, thus reducing the options from T to $T_h < T$.

GH1 says that the entire algorithm cannot contain duplicate actions (except the STOP action). GH2 allows to define the actions available in each event handler. For RBcast, we define that the agent can select all actions in both event handlers, except the DELIVER action on the *RB-Broadcast* event handler; as the *RB-Broadcast* event handler is only executed by one process, the DELIVER action must exist on the *receive* event handler, so that all processes can deliver the message, thus the possible existence of the DELIVER action on the *RB-Broadcast* event handler is redundant. GH3 allows to define the conditions available in each event handler. Based on this heuristic, we define that on the *receive* event handler, all considered conditions are allowed (see Section IV-A2). In the *RB-Broadcast*, we only allow conditions based on condition 0, since in that event handler the processes do not receive any message. GH4 allows to define that, for each condition and message type sent, the agent must choose between sending to all, to the neighbours, or only to itself. GH5 allows to define a message type for the first communication step of the algorithm (*RB-Broadcast* event handler). GH6 allows to restrict the size of the algorithm generated in terms of the number of actions for each event handler. As previously explained, we took inspiration from one of the most efficient RBcast algorithms [47], so we defined a minimum number of 2 and a maximum number of 4 actions in each event handler. GH7 forbids the agent to select actions based on invalid conditions, e.g., we forbid the agent to select actions that wait for message types not yet contained in the algorithm. GH8 forces the generation of algorithms with at least one DELIVER action, as that action clearly must exist

in the algorithm. GH9 allows to decrease the convergence time by discarding incorrect algorithms that are not related to the solution, which is equivalent to giving an infinite cost. GH10 allows to define the maximum number of types that the algorithm can contain – in this work, we defined only two possible types (cf. Section IV-A3).

V. RB-ORACLE

The RB-Oracle is the agent responsible for verifying the algorithms created by the RB-Learner agent, i.e., to implement the *verification process*. This section explains the verification process executed in the context of one episode.

The verification process is responsible for assessing the correctness of the algorithms generated, i.e., for assessing if each algorithm satisfies the RBcast properties (Section II-B) within one of the variants of the system model (Section II-A). Every episode, the RB-Learner generates an algorithm, and the RB-Oracle verifies it.

Automatic verification of a fault-tolerant distributed algorithm can be achieved using different techniques such as *model checking* [23], [49] or *theorem proving* [66]. In this paper, we use a model checking tool called *Spin* [46], a widely used framework on the verification of fault-tolerant algorithms [19], [23], [49], [62] that allows to build models and verify them.

Spin supports a few modes. We use Spin in *simulation* mode, i.e., we use it to simulate the execution of the created algorithm in a specific system model, doing an exhaustive exploration of the state space. In essence, during the state space exploration, Spin verifies if none of the three RBcast properties (RB-Agreement, RB-Validity, and RB-Integrity) is violated. The three properties, the protocol, the values of N and F, the system architecture, the behavior of each failure mode (No/Crash/Byzantine-Failure mode), the process that initiates the verification (randomly selected), and the faulty processes (also randomly selected, for $F > 0$) are all specified in PROMELA (Process or Protocol Meta Language). This is achieved by creating a system model in a PROMELA language file (*.pml* extension). Then, based on the *.pml* file, RB-Oracle builds a verification file (*pan.c*) – a *C* program that performs a verification of the correctness requirements for the system – and compiles it using *gcc*, generating an executable file. Lastly, the agent uses Spin to run the executable file and, with that, check the correctness of the algorithm.

The RB-Oracle verifies algorithms considering three failure modes: No-Failure, Crash-Failure, and Byzantine-Failure.

For the *No-Failure mode*, the RB-Oracle verifies the algorithm considering that all processes are correct, i.e., by following the actions of the algorithm without deviations. In this mode, in the experiments, we assume a system with $N = 3$ processes and $F = 0$. Moreover, we model only one possible verification of the system: since the algorithm that runs in each process is the same, more than one model would be redundant.

TABLE 5. Number of lines of code and programming language used.

Module	Lines of Code	Programming Language
Client	217	Python3
Generator (containing the RB-Learner)	2739	Python3
Verifier (containing the RB-Oracle)	2284	Python3 and PROMELA

In the *Crash-Failure mode*, we simulate the crash of the process assuming the worst case possible: crash failures happen between the sending of messages since the impact of a crash failure is the highest when it leads a message to be delivered only to a fraction of the processes. In this mode, in the experiments, we assume a system with $F = 1$ faulty processes and $N = 3$ processes, the minimum necessary to have $F = 1$ failures (see Section II-A). Moreover, for this mode, we build two models: when the process that initiates the algorithm is correct and when it is faulty. This allows verifying cases when either of the event handlers fail.

For the *Byzantine-Failure mode*, the RB-Oracle models a range of attacks where all faulty processes send the same malicious message to a predefined group of correct processes – from a group with 0 processes, and consequently, not sending to anyone, to sending to $N - F$ processes, and consequently, sending to all correct processes. In this mode, in the experiments, we assume a system with $F = 1$ faulty processes and $N = 4$ processes, which is the minimum number of processes necessary to have $F = 1$ faulty processes (see Section II-A). Moreover, similar to the process on the Crash-Failure mode, in this mode we also build two models: one to model a failure on each event handler of the algorithm.

VI. IMPLEMENTATION

As explained in the previous sections, our implementation[4] is based on two agents: RB-Learner and RB-Oracle. Table 5 shows the number of lines of code and the programming language used to implement the entire solution. The *Client* component is where the user defines the inputs and starts the execution of the solution. The *Generator* component is the one responsible for the generation of the algorithms, containing the RB-Learner. RB-Learner uses the Q-Learning algorithm and the UCB policy to implement the generation and learning of the algorithms. The *Verifier* component is the one that contains the RB-Oracle, responsible for the verification of the algorithms. RB-Oracle models a distributed system with N processes and F faulty processes, simulating the execution of the algorithm generated using the Spin framework. RB-Learner was implemented in Python3. RB-Oracle was implemented using both Python3, PROMELA (a verification modeling language used for Spin), and the Spin model checker.

VII. EXPERIMENTAL EVALUATION

Our experimental evaluation aims to assess the effectiveness and correctness of our solution. It will answer the following questions:

[4] https://diogolvaz.github.io/FAULTAGE/

TABLE 6. Experimental evaluation inputs.

Simulation Process inputs	Value
Number of simulations	5
Number of episodes	12.000
Generation Process inputs	**Value**
Rewards	Defined at Table 3
Heuristics to be applied and configurations associated	Configured as presented on Section IV-E
Verification Process inputs	**Value**
Failure modes to test	No-Failure, Crash-Failure and Byzantine-Failure modes
System Models to use	$[N = 3, F = 0]$ for No-Failure, $[N = 3, F = 1]$ for Crash-Failure and $[N = 4, F = 1]$for Byzantine-Failure
Properties to verify	*RB-Agreement*, *RB-Validity* and *RB-Integrity*

1) How many states does the agent explore until the first correct algorithm and the most efficient algorithm are found?
2) How many algorithms are generated in total in each experiment?
3) How many algorithms are generated until the first correct algorithm is generated?
4) What is the proportion of correct and incorrect algorithms from the total number of generated algorithms?
5) How does each proposed heuristic influence the learning process?
6) Is the agent able to generate a new algorithm, i.e., an algorithm that as far as we know has not been generated by humans?

We ran our experiments on a single machine with 32 vCPUs, 64 GB of memory, and Debian 10. We made experiments for three cases: (1) No-Failure mode; (2) Crash-Failure mode ; and (3) Byzantine-Failure mode. All results shown in the next sections are, except when noticed, averages of 5 simulations runs, each with $12,000$ episodes – the minimum number of episodes that we have found to be possible for the agent to converge to the most efficient algorithms in all experiments. The $12,000$ episodes took ± 9 hours to run on the No-Failure experiment, ± 3 days to run on the Crash-Failure experiment, and ± 7 days to run on the Byzantine-Failure experiment. This increase in time is due to the time needed for Spin to verify the models. Table 6 summarizes the inputs considered for the experimental evaluation.

A. STATES EXPLORED

For each algorithm generated, the agent explores multiple states when selecting the actions. This first set of experiments assesses the number of states explored in each experiment. Figure 3 shows the total number of states explored by the agent for each episode, on the entire experiment. Table 7 shows the number of states explored until the agent generated the first correct algorithm.

As expected, we can see in both figures that the agent needs to explore more states when the complexity of the problem to solve increases, i.e., the agent needs to explore more states when trying to find a Byzantine-tolerant algorithm – almost $20,000$ states – when compared to a Crash-tolerant or a non-fault-tolerant algorithm – around $12,000$ and $3,000$ states, respectively. Additionally, the agent also takes more time to converge when trying to find a Byzantine-tolerant algorithm – between $8,000$ and $10,000$ episodes – when compared to the other cases – between $1,000$ and $2,000$ episodes for the No-Failure algorithm and between $4,000$ and $6,000$ episodes for the Crash-Failure algorithm.

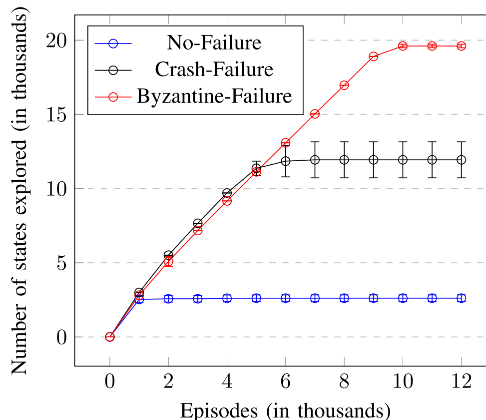

FIGURE 3. Number of states explored during each experiment.

TABLE 7. Number of states generated until the first correct algorithm is generated.

Experiment	States
No-Failure	15.4 ± 0.0
Crash-Failure	33.4 ± 16.9
Byzantine-Failure	13462.8 ± 6909.4

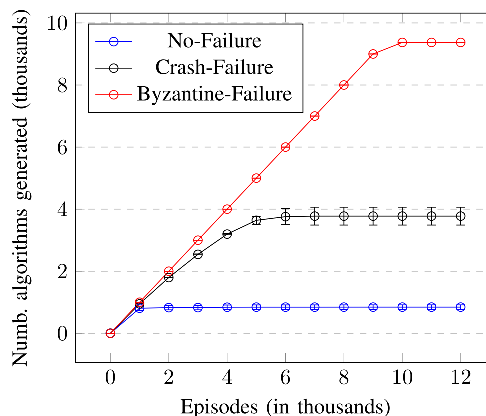

FIGURE 4. Number of algorithms generated during each experiment.

B. ALGORITHMS GENERATED

The agent generates multiple algorithms with the objective of learning from them. Therefore, we decided to assess how many algorithms the agent generates in each experiment. Figure 4 shows the number of algorithms generated by the agent per episode. Table 8 shows the number of correct

TABLE 8. Number of correct and incorrect algorithms generated in each experiment and number of algorithms generated until the first correct algorithm.

Experiment	Correct Algo.	Incorrect Algo.	Number Algo.
No-Fail.	341.8 ± 23.2	502.0 ± 53.3	2.6 ± 1.9
Crash-Fail.	420.2 ± 32.1	3355.6 ± 256.6	6.8 ± 4.1
Byzantine-Fail.	3.0 ± 0.0	9368.6 ± 6.8	6260.0 ± 3467.9

and incorrect algorithms generated, as also the number of algorithms generated until the first correct algorithm. As expected, and similarly to what happens with the number of states, the agent needs to generate more algorithms, as also takes more time to converge, with the increase of the complexity of the problem to solve. Moreover, another interesting aspect is the percentage of incorrect algorithms generated by the agent in each test: $\pm 60\%$ on the No-Failure test, $\pm 89\%$ on the Crash-Failure test, and 99.9% on the Byzantine-Failure test, which means that, even with all the Heuristics defined, the agent still has a difficult task generating a correct algorithm.

Algorithm 1 Most Efficient RBcast Algorithm for a No-Failure Experiment Generated by the RB-Learner

1: *when RB-Broadcast(m) do*:
2: `SEND to all`(< *type0*,m>) if *true* and not already sent;
3: `STOP` if *true*;
4: *when receive(< t,m >) do*:
5: `DELIVER`(< m>) if *true* and not already delivered;
6: `STOP` if *true*;

In the No-Failure mode, the agent converged to Algorithm 1 in 4 of the simulations executed. This algorithm is equivalent to one presented in [11]: both exchange, at most, N messages, require 1 communication step and 1 message type (or none), and need to receive 1 message.

On the Crash-Failure mode, the agent also converged to Algorithm 2 in 4 of the simulations executed. Note that the algorithm sends a new message type on the *receive* event handler (*type1*) when it could send the *type0*. This happens because of the heuristic GH5, which only allows sending messages of *type0* on the *RB-Broadcast* event handler. This algorithm is similar to the one presented in [26] and [67]: both exchange, at most, $N^2 - N + 1$ messages, require 1 communication step and 1 message type (or none) and need to receive 1 message.

On the Byzantine-Failure mode, Algorithm 3 is one of the algorithms generated by the agent in all the simulations executed. This algorithm is one of the most efficient algorithms developed and it is similar to the one presented in [47]: both exchange, at most, $N^2 + N$ messages, require 2 communication steps and 2 message types, and need to receive $(N + F)/2$ messages. However, in this experiment, the agent converged to a new efficient algorithm discussed in Section VII-D

Algorithm 2 Most Efficient RBcast Algorithm for a Crash-Failure Experiment Generated by the RB-Learner

1: *when RB-Broadcast(m) do*:
2: `SEND to myself`(< *type0*,m>) if *true* and not already sent;
3: `STOP` if *true*;
4: *when receive(< t,m >) do*:
5: `SEND to neighbours`(< *type1*,m>) if *true* and not already sent;
6: `DELIVER`(< m>) if *true* and not already delivered;
7: `STOP` if *true*;

Algorithm 3 Most Efficient RBcast Algorithm for a Byzantine-Failure Experiment Generated by the RB-Learner

1: *when RB-Broadcast(m) do*:
2: `SEND to all`(< *type0*,m>) if *true* and not already sent;
3: `STOP` if *true*;
4: *when receive(< t,m >) do*:
5: `SEND to all`(< *type1*,m>) if received (< *type0*,m>) from 1 distinct party and not already sent;
6: `DELIVER`(< m>) if received (< *type1*,m>) from $(N + F)/2$ distinct parties and not already delivered;
7: `SEND to all`(< *type1*,m>) if received (< *type1*,m>) from $F + 1$ distinct parties and not already sent;
8: `STOP` if *true*;

C. IMPACT OF THE HEURISTICS

The heuristics we defined (see Section IV-E) guide the agent by helping it to avoid incorrect algorithms. They do not help to obtain algorithms or algorithms that are more correct but reduce the number of states to explore. In this evaluation, we analyzed the importance of each heuristic with the Crash-Failure experiment.

To achieve this, we ran one experiment with each GH turned off and all others turned on. There were two exceptions. In GH6, we increased the maximum number of actions in each event from 4 to 5, but did not turn this heuristic off, to avoid the agent of generating algorithms with too many actions. For GH10, we increased the maximum number of types from 2 to 3 but did not turn it off, as the agent could explore too many types. We executed one simulation with 10, 000 episodes for each experiment.

Figure 5 shows the evolution of the number of algorithms generated with each GH turned off, from where we conclude that all heuristics are important to reduce the number of states explored until a correct and efficient algorithm is obtained.

D. A NEW BYZANTINE-TOLERANT ALGORITHM

This section answers the last question, i.e., the possibility of our agent finding new algorithms. On the Byzantine-Failure simulations, the agent converged to Algorithm 4, a new

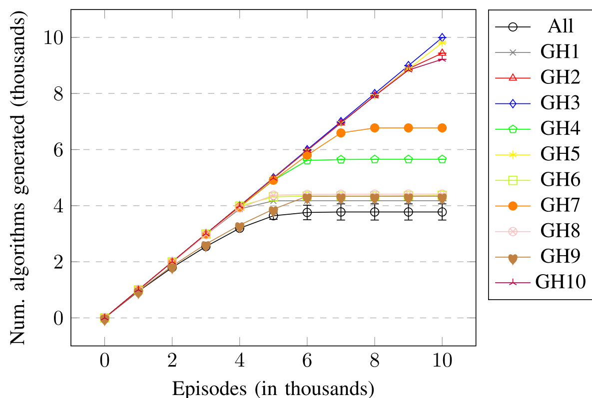

FIGURE 5. Number of algorithms generated during each experiment without the identified GH. The All line represents an experiment with all GH turned on.

Algorithm 4 New Byzantine-Tolerant Algorithm Generated by the RB-Leaner

1: *when RB-Broadcast(m) do*:
2: `SEND to neighbours`(< *type0*,m>) if *true* and not already sent;
3: `STOP` if *true*;
4: *when receive(< t,m >) do*:
5: `SEND to neighbours`(< *type1*,m>) if received (< *type0*,m>) from 1 distinct parties and not already sent;
6: `SEND to neighbours`(< *type1*,m>) if received (< *type1*,m>) from $F+1$ distinct parties and not already sent;
7: `DELIVER`(< m>) if received (< *type1*,m>) from $F+1$ distinct parties and not already delivered;
8: `STOP` if *true*;

and efficient Byzantine-tolerant algorithm for $F = 1$ and $N \geq 4 \in \mathbb{N}$.

When comparing the efficiency of Algorithms 3 and 4, we see two improvements on Algorithm 4: (1) Algorithm 3 instructs processes to send messages to all, meaning a total of N^2+N messages sent, while Algorithm 4 instructs processes to send messages only to the neighbours, saving the cost of processes sending a message to themselves and meaning a total of $(N-1)+N(N-1)$ messages sent. Table 9 summarizes the total number of messages sent by each algorithm, assuming different system configurations; (2) Algorithm 3 needs $(N+F)/2$ messages of *type1* to deliver a message, while Algorithm 4 only needs $F+1$ messages of the same type to deliver. Since, for $F = 1$ and $N \geq 4 \in \mathbb{N}$ we have that $(N+F)/2 > F+1$, then Algorithm 4 is more efficient from the message delivery point of view. More precisely, the new generated Algorithm 4 will always require $F+1 = 1+1 = 2$ messages, while on Algorithm 3, the number of messages needed increases with the increase of the total number N of processes, since threshold $(N+F)/2$ depends on N.

TABLE 9. Total number of messages during the execution of both algorithms.

N & F	Algorithm	Total number of messages sent
$N = 4$ & $F = 1$	3	$4^2 + 4 = 20$
	4	$(4-1) + 4 * (4-1) = 15$
$N = 10$ & $F = 1$	3	$10^2 + 10 = 110$
	4	$(10-1) + 10(10-1) = 99$
$N = 100$ & $F = 1$	3	$100^2 + 100 = 10100$
	4	$(100-1) + 100(100-1) = 9999$
$N = 1000$ & $F = 1$	3	$1000^2 + 1000 = 1001000$
	4	$(1000-1) + 1000(1000-1) = 999999$

This result shows the capability of RB-Learner to adapt to the specified problem with modifications and learn to generate efficient algorithms for that case also. Moreover, reinforces the possibility of this approach to help develop new distributed fault-tolerant algorithms, allowing to advance the state-of-the-art in the distributed computing field.

VIII. RELATED WORK

Fault-tolerant algorithms have been widely studied over the years [6], [8], [9], [12], [13], [16], [17], [33], [47], [48], [60], [67]. These algorithms: solve different problems, such as *Reliable Broadcast* [47], *Consensus* [12] or *Leader Election* [63]; tolerate different failure modes, like *Crash* [26] and *Byzantine* [17]; use different communication models, namely *fully-connected* [16] and *partially-connected* [8]; and tolerate different fault ratios, such as $\lfloor (N-1)/3 \rfloor$ [9] and $\lfloor (N-1)/2 \rfloor$ [17], where N is the number of components in the system. However, as far as we know, all works are based on manual processes.

Program Synthesis [29], [35] is the task of automatically discovering and developing programs that satisfy requirements expressed in some form of specification by a user. In this work, our goal is to develop a tool capable of developing *algorithms* – in pseudo-code – that, later, can be implemented by a *program* – in a specific programming language. Program synthesis has been applied to generate security protocols [69] , control plane operations [27] or switch code [32]. Applied to the field of distributed algorithms, we identified seven works: three focused on the generation and discovery of mutual exclusion algorithms [4], [36], [37], two that automatically generate consensus algorithms [30], [74], another that synthesizes fault-tolerant distributed algorithms [56], and one focused on the synthesis of leader election algorithms [38]. However, all previous work is based on techniques different from machine learning, such as brute-force approaches that generate all possible algorithms [4], [30], [74] or solutions using genetic programming [36], [37], [38]. Moreover, a majority focus on shared memory systems [4], [30], [36], [37] and some only consider the correctness of the algorithms generated [4], [30], [56]. In this work, we present a novel solution that generates correct and efficient Reliable Broadcast using a machine learning technique – Reinforcement Learning – so

that the software can *learn* to generate fault-tolerant distributed algorithms.

Recently, supervised machine learning techniques have been used to automatically generate local and non-distributed code [3], [18], [58], [68]. In this work, we propose the use of a different machine learning technique – *Reinforcement Learning* – to generate fault-tolerant distributed code. Based on this technique, we have found a work that presents a framework to improve pre-trained language models for program synthesis tasks through deep reinforcement learning [57], a work that uses reinforcement learning to generate matrix multiplication algorithms [22], a work that uses reinforcement learning to generate tests for Android GUI applications [39], and a work that uses reinforcement learning coupled with deductive reasoning to program synthesis [28]. However, unlike ours, the first work uses deep reinforcement learning only for the optimization of the pre-trained language models, and the others are focused on a problem very different from ours.

Program Verification [31], [44] is the process of ensuring that a given program behaves as intended and meets its specified requirements, helping to ensure the correctness, reliability, and security of the programs. For the verification of fault-tolerant algorithms [23], [24], [53], [66], we have identified and used the Spin/PROMELA [46] model checker [23], [49] framework. This decision is based on the fact that Spin is a very flexible tool that allows modeling different fault-tolerant distributed algorithms such as Consensus [71], Reliable Broadcast [49] or Leader Election [1]. Moreover, Spin is a very mature and robust framework with extensive documentation and an active community.[5]

IX. CONCLUSION

Fault-tolerant algorithms have been studied over the years, discussing different problems and variants. However, this study is complex and has always been based on human-based processes. To automate such processes, we propose a novel solution based on a machine learning technique. We present a first implementation of the approach based on two agents, RB-Learner and RB-Oracle, capable of learning to synthesize the RBcast algorithm. As we have presented during the experimental evaluation, our solution can synthesize correct and efficient algorithms, depending on the properties of the problem, proving the effectiveness of the proposed approach in solving distributed problems when compared with the manual process. To our knowledge, this work is the first that merges both areas of generation and verification into an automatic process capable of generating correct and efficient RBcast algorithms using machine learning techniques. For further research, we aim to apply our approach to different distributed problems, like *Consensus*, and try to decrease the number of inputs needed, to further decouple our agent from knowledge based on previous works, e.g. the threshold expressions.

[5]https://spinroot.com/spin/whatispin.html

REFERENCES

[1] X. An and J. Pang, "Model checking round-based distributed algorithms," in *Proc. 15th IEEE Int. Conf. Eng. Complex Comput. Syst.*, Mar. 2010, pp. 127–135.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] B. Asiroglu, B. R. Mete, E. Yildiz, Y. Nalçakan, A. Sezen, M. Dagtekin, and T. Ensari, "Automatic HTML code generation from mock-up images using machine learning techniques," in *Proc. Sci. Meeting Elect.-Electron. Biomed. Eng. Comput. Sci. (EBBT)*, Apr. 2019, pp. 1–4.

[4] Y. Bar-David and G. Taubenfeld, "Automatic discovery of mutual exclusion algorithms," in *Proc. Int. Symp. Distrib. Comput.* Cham, Switzerland: Springer, 2003, pp. 136–150.

[5] R. A. Bazzi and G. Neiger, "Simplifying fault-tolerance: Providing the abstraction of crash failures," *J. ACM*, vol. 48, no. 3, pp. 499–554, May 2001.

[6] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proc. 2nd Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, 1983, pp. 27–30.

[7] W. Bibel, *Automated Theorem Proving*. Springer, 2013.

[8] S. Bonomi, J. Decouchant, G. Farina, V. Rahli, and S. Tixeuil, "Practical Byzantine reliable broadcast on partially connected networks," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2021, pp. 506–516.

[9] G. Bracha, "An asynchronous [(n-1)/3]-resilient consensus protocol," in *Proc. 3rd Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, 1984, pp. 154–162.

[10] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, Oct. 1985.

[11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[12] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd USENIX Symp. Operating Syst. Design Implement.*, 1999, pp. 173–186.

[13] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 251–273, Aug. 1984.

[14] A. P. L. Ferreira, "Model checking," in *Proc. Workshop-School Theor. Comput. Sci.* Cham, Switzerland: Springer, Aug. 2011, pp. 9–14.

[15] M. Coggan, "Exploration and exploitation in reinforcement learning," Res. Supervised Prof. Doina Precup, CRA-W DMP Project at McGill Univ., Montreal, QC, Canada, Tech. Rep., 2004.

[16] M. Correia, N. F. Neves, and P. Veríssimo, "From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures," *Comput. J.*, vol. 49, no. 1, pp. 82–96, Jan. 2006.

[17] M. Correia, G. S. Veronese, and L. C. Lung, "Asynchronous Byzantine consensus with 2f+1 processes," in *Proc. ACM Symp. Appl. Comput.*, Mar. 2010, pp. 475–480.

[18] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code generation using machine learning: A systematic review," *IEEE Access*, vol. 10, pp. 82434–82455, 2022.

[19] G. Delzanno, M. Tatarek, and R. Traverso, "Model checking Paxos in spin," *Electron. Proc. Theor. Comput. Sci.*, vol. 161, pp. 131–146, Aug. 2014.

[20] C. Dwork and Y. Moses, "Knowledge and common knowledge in a Byzantine environment: Crash failures," *Inf. Comput.*, vol. 88, no. 2, pp. 156–186, Oct. 1990.

[21] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," *ACM Trans. Comput. Syst.*, vol. 21, no. 4, pp. 341–374, Nov. 2003.

[22] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, Oct. 2022.

[23] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Tutorial on parameterized model checking of fault-tolerant distributed algorithms," in *Proc. Int. School Formal Methods Design Comput., Commun. Softw. Syst.* Cham, Switzerland: Springer, 2014, pp. 122–171.

[24] A. Goel and K. A. Sakallah, "Towards an automatic proof of Lamport's Paxos," in *Formal Methods Computer-Aided Design*. Oct. 2021, pp. 112–122.

[25] Boston Consulting Group. *Generative AI*. Accessed: Jan. 2023. [Online]. Available: https://www.bcg.com/x/artificial-intelligence/generative-ai

[26] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell Univ., Dept. Comput. Sci., Ithaca, NY, USA, Tech. Rep., TR94-1425, May 1994.

[27] E. H. Campbell, W. T. Hallahan, P. Srikumar, C. Cascone, J. Liu, V. Ramamurthy, H. Hojjat, R. Piskac, R. Soulé, and N. Foster, "Avenir: Managing data plane diversity with control plane synthesis," in *Proc. NSDI*, 2021, pp. 133–153.

[28] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, "Program synthesis using deduction-guided reinforcement learning," in *Proc. 32nd Int. Conf. Comput. Aided Verification*. Los Angeles, CA, USA: Springer, Jul. 2020, pp. 587–610.

[29] C. David and D. Kroening, "Program synthesis: Challenges and opportunities," *Phil. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 375, no. 2104, Oct. 2017, Art. no. 20150403.

[30] C. Delporte-Gallet, H. Fauconnier, Y. Jurski, F. Laroussinie, and A. Sangnier, "Towards synthesis of distributed algorithms with SMT solvers," in *Proc. Networked Syst., 7th Int. Conf.* Marrakech, Morocco: Springer, Jun. 2019, pp. 200–216.

[31] J. H. Fetzer, "Program verification: The very idea," *Commun. ACM*, vol. 31, no. 9, pp. 1048–1063, Sep. 1988.

[32] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, "Switch code generation using program synthesis," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 44–61.

[33] G. Zhang, F. Pan, M. Dang'ana, Y. Mao, S. Motepalli, S. Zhang, and H.-A. Jacobsen, "Reaching consensus in the Byzantine Empire: A comprehensive review of BFT consensus algorithms, 2022.

[34] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Commun. ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[35] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, nos. 1–2, pp. 1–119, 2017.

[36] G. Katz and P. Doron, "Genetic programming and model checking: Synthesizing new mutual exclusion algorithms," in *Proc. Automated Technol. Verification Anal., 6th Int. Symp. (ATVA)*. Seoul, South Korea: Springer, Oct. 2008, pp. 33–47.

[37] G. Katz and D. Peled, "Model checking-based genetic programming with an application to mutual exclusion," in *Proc. Tools Algorithms Construct. Anal. Syst., 14th Int. Conf., (TACAS)*. Budapest, Hungary: Springer, Mar. 2008, pp. 141–156.

[38] G. Katz and D. Peled, "Synthesizing solutions to the leader election problem using model checking and genetic programming," in *Proc. Hardw. Software, Verification Test., 5th Int. Haifa Verification Conf.* Haifa, Israel: Springer, Oct. 2009, pp. 117–132.

[39] Y. Koroglu and A. Sen, "Reinforcement learning-driven test generation for Android GUI applications using formal specifications," 2019, *arXiv:1911.05403*.

[40] X. Liu, B. Farahani, and F. Firouzi, "Distributed ledger technology," in *Intelligent Internet of Things: From Device to Fog and Cloud*. Springer, 2020, pp. 393–431.

[41] McKinsey & Company. (Jan. 2023). *What is generative AI?* [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-generative-ai

[42] D. T. Patel, "Distributed computing for Internet of Things (IoT)," in *Research Anthology on Architectures, Frameworks, and Integration Strategies for Distributed and Cloud Computing*. Pennsylvania, PA, USA: IGI Global, 2021, pp. 1874–1894.

[43] M.-C. Roşu, "Early-stopping terminating reliable broadcast protocol for general-omission failures," in *Proc. 15th Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, 1996, p. 209.

[44] K. Sieber, *The Foundations of Program Verification*. New York, NY, USA: Springer-Verlag, 2013.

[45] *World Economic Forum & Visual Capitalist. What is generative AI? An AI explains*. (Feb. 2023). [Online]. Available: https://www.weforum.org/agenda/2023/02/generative-ai-explain-algorithms-work

[46] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[47] D. Imbs and M. Raynal, "Trading off t-resilience for efficiency in asynchronous Byzantine reliable broadcast," 2015, *arXiv:1510.06882*.

[48] D. Imbs and M. Raynal, "Trading off t-resilience for efficiency in asynchronous Byzantine reliable broadcast," *Parallel Process. Lett.*, vol. 26, no. 4, Dec. 2016, Art. no. 1650017.

[49] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Towards modeling and model checking fault-tolerant distributed algorithms," in *Int. SPIN Workshop Model Checking Softw.*, pp. 209–226. Springer, 2013.

[50] C. Johnen, L. Arantes, and P. Sens, "FIFO and atomic broadcast algorithms with bounded message size for dynamic systems," in *Proc. 40th Int. Symp. Reliable Distrib. Syst. (SRDS)*, Sep. 2021, pp. 277–287.

[51] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, no. 1, pp. 237–285, Jan. 1996.

[52] I. Konnov and J. Widder, "ByMC: Byzantine model checker," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*. Cham, Switzerland: Springer, 2018, pp. 327–342.

[53] L. Lamport and S. Merz, "Specifying and verifying fault-tolerant systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Cham, Switzerland: Springer, 1994, pp. 41–76.

[54] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[55] A. D. Laud, *Theory and Application of Reward Shaping in Reinforcement Learning*. Champaign, IL, USA: Univ. Illinois at Urbana-Champaign, 2004.

[56] M. Lazic, I. Konnov, J. Widder, and R. Bloem, "Synthesis of distributed algorithms with parameterized threshold guards," in *Proc. 21st Int. Conf. Princ. Distrib. Syst. (OPODIS)*. Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, pp. 1–20.

[57] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "CodeRL: Mastering code generation through pretrained models and deep reinforcement learning," 2022, *arXiv:2207.01780*.

[58] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surveys*, vol. 53, no. 3, pp. 1–38, May 2021.

[59] D. B. Lenat, "The nature of heuristics," *Artif. Intell.*, vol. 19, no. 2, pp. 189–249, Oct. 1982.

[60] J. Li, T. Yu, Y. Wang, and R. Wattenhofer, "Dynamic Byzantine broadcast in asynchronous message-passing systems," *IEEE Access*, vol. 10, pp. 91372–91384, 2022.

[61] N. A. Lynch, *Distributed Algorithms*. San Mateo, CA, USA: Morgan Kaufmann, 1996.

[62] T. Minamikawa, T. Tsuchiya, and T. Kikuno, "Language and tool support for model checking of fault-tolerant distributed algorithms," in *Proc. 14th IEEE Pacific Rim Int. Symp. Dependable Comput.*, Dec. 2008, pp. 40–47.

[63] D. Mitra, A. Cortesi, and N. Chaki, "ALEA: An anonymous leader election algorithm for synchronous distributed systems," in *Proc. Conf. Prog. Image Process., Pattern Recognit. Commun. Syst. (CORES, IP&C, ACS)*. Cham, Switzerland: Springer, 2022, pp. 46–58.

[64] H. Müller-Merbach, "Heuristics and their design: A survey," *Eur. J. Oper. Res.*, vol. 8, no. 1, pp. 1–23, Sep. 1981.

[65] M. E. Peck, "Blockchains: How they work and why they'll change the world," *IEEE Spectr.*, vol. 54, no. 10, pp. 26–35, Oct. 2017.

[66] V. Rahli, I. Vukotic, M. Völp, and P. Esteves-Verissimo, "Velisarios: Byzantine fault-tolerant protocols powered by Coq," in *Proc. Eur. Symp. Program.* Cham, Switzerland: Springer, 2018, pp. 619–650.

[67] M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*. Springer, 2018.

[68] J. Shin and J. Nam, "A survey of automatic code generation from natural language," *J. Inf. Process. Syst.*, vol. 17, no. 3, pp. 537–555, 2021.

[69] D. Song, A. Perrig, and D. Phan, "AGVI—Automatic generation, verification, and implementation of security protocols," in *Proc. Comput. Aided Verification, 13th Int. Conf.*, Paris, France: Springer, Jul. 2001, pp. 241–245.

[70] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[71] T. Tsuchiya and A. Schiper, "Using bounded model checking to verify consensus algorithms," in *Proc. Distrib. Comput. 22nd Int. Symp.* Arcachon, France: Springer, Sep. 2008, pp. 466–480.

[72] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, 1992.

[73] M. A. Wiering and M. Van Otterlo, ''Reinforcement learning,'' *Adaptation, Learn. Optim.*, vol. 12, no. 3, p. 729, 2012.

[74] P. Zielinski, ''Automatic verification and discovery of Byzantine consensus protocols,'' in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2007, pp. 72–81.

DIOGO VAZ (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science and engineering from Instituto Superior Técnico (IST), University of Lisbon, Portugal, in 2019 and 2021, respectively. He is currently pursuing the Ph.D. degree with IST. He is a Junior Researcher with INESC-ID Lisboa. His current research interests include cybersecurity and distributed systems, mainly in the application of machine learning in both areas.

DAVID R. MATOS (Member, IEEE) received the B.Sc. and M.Sc. degrees in informatics engineering from the Faculty of Sciences, University of Lisbon, in 2012 and 2013, respectively, and the Ph.D. degree in computer sciences and engineering from Instituto Superior Técnico, University of Lisbon, in 2019. He is currently an Invited Assistant Professor at Instituto Superior Técnico, University of Lisbon. His current research interests include distributed systems and cybersecurity.

MIGUEL L. PARDAL (Member, IEEE) received the bachelor's, master's, and Ph.D. degrees in computer science and engineering from Instituto Superior Técnico (IST), University of Lisbon, Portugal, in 2000, 2006, and 2014, respectively. He was a Guest Scientist with the Chair of Network Architectures and Services, TU Munich. During his Ph.D. degree, he was a Visiting Student with the Auto-ID Labs, MIT. He is currently an Assistant Professor with IST and a Researcher with the Distributed, Parallel and Secure Systems Group (DPSS), INESC-ID. He has led the SureThing Project (FCT) and completed participation in the Safe Cloud EU Project (H2020). His current research interests include cybersecurity applied to the digital frontiers of the Internet of Things and cloud computing.

MIGUEL CORREIA (Senior Member, IEEE) is currently a Full Professor with the Computer Science and Engineering Department (DEI), Instituto Superior Técnico (IST), University of Lisbon (ULisbon), Lisbon, Portugal. He is also a Senior Researcher with INESC-ID. He has been involved in many international and national research projects related to distributed systems, including the TRUSTyFOOD, DE4A, BIG, QualiChain, SPARTA, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 200 publications. His current research interests include cybersecurity and fault tolerance, typically in distributed systems, and the context of different applications (blockchain, cloud, and mobile). He is a member of the board.