Correctness-by-learning of infinite-state component-based systems

Haitham Bou-Ammar, Mohamad Jaber, Mohamad Nassar

American University of Beirut, Beirut, Lebanon
{hb71, mj54, mn115}@aub.edu.lb

Abstract. We introduce a novel framework for runtime enforcement of safe executions in component-based systems with multi-party interactions modeled using BIP. Our technique frames runtime enforcement as a sequential decision making problem and presents two alternatives for learning optimal strategies that ensure fairness between correct traces. We target both finite and infinite state-spaces. In the finite case, we guarantee that the system avoids bad-states by casting the learning process as a one of determining a fixed point solution that converges to the optimal strategy. Though successful, this technique fails to generalize to the infinite case due to need for building a dictionary, which quantifies the performance of each state-interaction pair. As such, we further contribute by generalizing our framework to support the infinite setting. Here, we adapt ideas from function approximators and machine learning to encode each state-interaction pairs' performance. In essence, we autonomously learn to abstract similar performing states in a relevant continuous space through the usage of deep learning. We assess our method empirically by presenting a fully implemented tool, so called `RERL`. Particularly, we use `RERL` to: 1) enforce deadlock freedom on a dining philosophers benchmark, and 2) allow for pair-wise synchronized robots to autonomously achieve consensus within a cooperative multi-agent setting.

1 Introduction

Building correct and efficient software systems in a timely manner is still a very challenging task despite the existence of a plethora of techniques and methods. For instance, correctness can be ensured using static analysis such as model checking [5,6,19] or dynamic analysis such as runtime verification [8]. Static analysis mainly suffers from state-space explosion whereas dynamic analysis suffers from its accuracy (reachability cover) and efficiency. To overcome the problem of state-space explosion, abstraction techniques [9] can be used, however, it has the effect of false negatives. Moreover, software synthesis, correct-by-design, was introduced to automatically generate implementation from high-level designs. However, correct-by-design was proven to be NP-hard [17] in some cases and undecidabile [18] in some main classical automatic synthesis problems. On the other hand, developing implementations that are compliant with their specifications require a careful attention from designers and developers. Can we relax the

development process by giving the option to over-approximate the behaviors of the implementations, i.e., introduce additional behaviors w.r.t. the given specification? This relaxation would drastically simplify the development process, though it may introduce errors.

In this paper, we introduce a new runtime enforcement technique that takes a software system with extra behaviors (w.r.t. a specification) and uses static and dynamic techniques with the help of machine learning to synthesize more accurate and precise behavior, i.e., remove the extra ones w.r.t. the given specification. We apply our method to component-based systems with multi-party interactions modeled using BIP [1]. BIP (Behavior, Interaction and Priority) allows to build complex systems by coordinating the behavior of atomic components. BIP has a rigorous operational semantics: the behavior of a composite component is formally described as the composition of the behaviors of its atomic components. From a given state of the components, the operational semantics define the next allowed interactions and their corresponding next states. D-Finder [2] is used to verify the correctness of BIP systems. While D-Finder uses compositional and abstraction techniques, it suffers from state-space explosion and producing false negatives. Dynamic analysis techniques [7,4] are also proposed for BIP systems. However, they only support a limited level of recovery. A detailed comparison is discussed in the related work.

Our technique frames runtime enforcement as a sequential decision making problem and presents two alternatives for learning optimal strategies that ensure fairness between correct traces. That is, the policy should not avoid correct traces from execution. We target both finite and infinite state-spaces. In the finite case, we guarantee that the system avoids bad-states by casting the learning process as a one of determining a fixed point solution that converges to the optimal strategy. Though successful, this technique fails to generalize to the infinite case due to need for building a dictionary, which quantifies the performance of each state-interaction pair, i.e., reduce the non-determinism by only allowing interactions leading to states that conform with the specifications. As such, we further contribute by generalizing our framework to support the infinite setting. Here, we adapt ideas from function approximators and machine learning to encode each state-interaction pairs' performance. In essence, we autonomously learn to abstract similar performing states in a relevant continuous space through the usage of deep learning. We assess our method empirically by presenting a fully implemented version called `RERL`. Particularly, we use `RERL` to: 1) enforce deadlock freedom on a dining philosophers benchmark, and 2) allow for pair-wise synchronized robots to autonomously achieve a consensus within a cooperative multi-agent setting.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 recall the necessary concepts of the BIP framework. Section 4 presents our main contribution, a runtime enforcement framework for component-based systems (finite and infinite state-space) using machine learning. Section 5 describes `RERL`, a full implementation of our framework and its

evaluation using two benchmarks. Section 6 draws some conclusions and perspectives.

2 Related work

Runtime enforcement of component-based systems has been introduced in [4] to ensure the correct runtime behavior (w.r.t. a formal specification) of a system. The authors define series of transformations to instrument a component-based system described in the BIP framework. The instrumented system allows to observe and avoid any error in the behavior of the system. The proposed method was fully implemented in RE-BIP. Although, contrarily to our method, the proposed method is sound (i.e., it always avoids bad states), it mainly suffers from two limitations. First, it only considers a 1-step recovery. That is, if the system enters a correct state from which all the reachable states are bad states, the method fails. Second, the instrumented system introduces a huge overhead w.r.t. original behavior. This overhead would be drastically increased to support more than 1-step recovery.

In [15,16], the authors introduced a predictive runtime enforcement framework that allows to build an enforcement monitor with or without a-priori knowledge of the system. The enforcement monitor ensures that the system complies with a certain property, by delaying or modifying events. The proposed method is theoretical and cannot be applied to real software systems as delaying or modifying events would require an infinite memory and also is not practical in software systems.

In [10], the authors proposed a game-theoretic method for synthesizing control strategies to maximize the resilience of software systems. The method allows the system to not take transition leading to bad states using game-theoretic method. Consequently, similar to RE-BIP, the proposed approach only allows a 1-step recovery. In other words, they need to do a back propagation from the bad states to re-label all good states as bad states when all their corresponding traces would lead to bad states, which is not feasible in case of infinite-state system.

Recent work [14,11,13] establishes techniques to synthesize code using genetic programming. In particular, the method randomly generates an initial population of programs based on a given configuration and then they apply mutation functions to optimize a given fitness function (w.r.t. specification). Nonetheless, the method was applied to communication protocols without reporting success rates. Moreover, deep learning is much more expressive than genetic programming, which failed to learn complex structures. Moreover, it is not clear how to automatically derive a fitness function from a given specification.

3 Behavior, Interaction and Priority (BIP)

We recall the necessary concepts of the BIP framework [1]. BIP allows to construct systems by superposing three layers of design: Behavior, Interaction, and

Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. The *interaction* layer provides the collaboration between components. Interactions are described using sets of ports. The *priority* layer is used to specify scheduling policies applied to the interaction layer, given by a strict partial order on interactions.

3.1 Atomic Components

We define *atomic components* as transition systems with a set of ports labeling individual transitions. These ports are used for communication between different components.

Definition 1 (Atomic Component). *An* atomic component B *is a labeled transition system represented by a triple* $(Q, P, \rightarrow)$ *where* Q *is a set of* states*,* P *is a set of* communication ports*,* $\rightarrow \subseteq Q \times P \times Q$ *is a set of* possible transitions*, each labeled by some port.*

For any pair of states $q, q' \in Q$ and a port $p \in P$, we write $q \xrightarrow{p} q'$, iff $(q, p, q') \in \rightarrow$. When the communication port is irrelevant, we simply write $q \rightarrow q'$. Similarly, $q \xrightarrow{p}$ means that there exists $q' \in Q$ such that $q \xrightarrow{p} q'$. In this case, we say that p is *enabled* in state q.

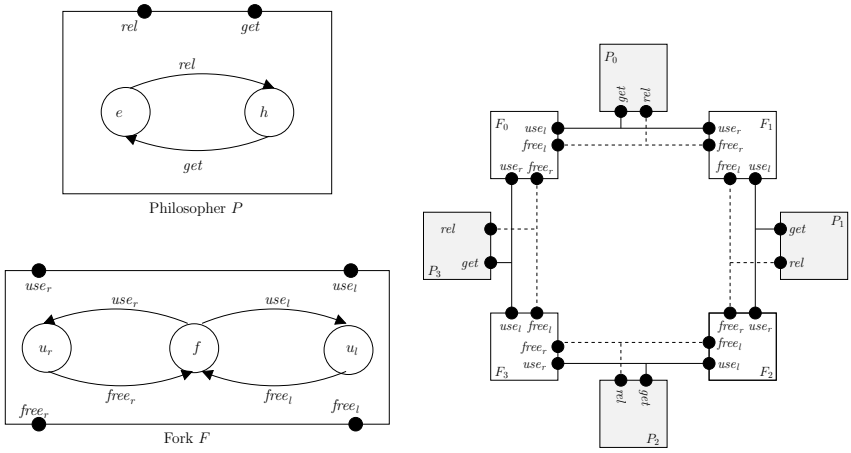
In practice, atomic components are extended with variables. Each variable may be bound to a port and modified through interactions involving this port. We also associate a guard and an update function (i.e., action) to each transition. A guard is a predicate on variables that must be true to allow the execution of the transition. An update function is a local computation triggered by the transition that modifies the variables.

Figure 1(a) shows an atomic component P that corresponds to the behavior of a philosopher in the dining-philosopher problem, where $Q = \{e, h\}$ denotes eating and hungry, $P = \{rel, get\}$ denotes releasing and getting of forks, and $\rightarrow = \{e \xrightarrow{rel} h, h \xrightarrow{get} e\}$.

3.2 Composition Component

For a given system built from a set of n atomic components $\{B_i = (Q_i, P_i, \rightarrow_i)\}_{i=1}^{n}$, we assume that their respective sets of ports are pairwise disjoint, i.e., for any two $i \neq j$ from $\{1..n\}$, we have $P_i \cap P_j = \emptyset$. We can therefore define the set $P = \bigcup_{i=1}^{n} P_i$ of all ports in the system. An *interaction* is a set $a \subseteq P$ of ports. When we write $a = \{p_i\}_{i \in I}$, we suppose that for $i \in I$, $p_i \in P_i$, where $I \subseteq \{1..n\}$.

Similar to atomic components, BIP extends interactions by associating a guard and a transfer function to each of them. Both the guard and the function are defined over the variables that are bound to the ports of the interaction. The guard must be true to allow the interaction. When the interaction takes place, the associated transfer function is called and modifies the variables.



(a) Philosopher P and fork F atomic components.

(b) Dining philosophers composite component with four philosophers.

Fig. 1. Dining philosophers.

Definition 2 (Composite Component). *A* composite component *(or simply* component*) is defined by a composition operator parameterized by a set of interactions* $\Gamma \subseteq 2^P$*.* $B \stackrel{def}{=} \Gamma(B_1, \ldots, B_n)$*, is a transition system* $(Q, \Gamma, \rightarrow)$*, where* $Q = \bigotimes_{i=1}^{n} Q_i$ *and* $\rightarrow$ *is the least set of transitions satisfying the following rule:*

$$\frac{a = \{p_i\}_{i \in I} \in \Gamma \qquad \forall i \in I : q_i \stackrel{p_i}{\rightarrow}_i q_i' \qquad \forall i \notin I : q_i = q_i'}{\boldsymbol{q} = (q_1, \ldots, q_n) \stackrel{a}{\rightarrow} \boldsymbol{q'} = (q_1', \ldots, q_n')}$$

The inference rule says that a composite component $B = \Gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \Gamma$, iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labeled with p_i; the states of components that do not participate in the interaction stay unchanged.

Figure 1(b) illustrates a composite component $\Gamma(P_0, P_1, P_2, P_3, F_0, F_1, F_2, F_3)$, where each P_i (resp. F_i) is identical to component P (resp. F) in Figure 1(a) and $\Gamma = \Gamma_{get} \cup \Gamma_{rel}$, where $\Gamma_{get} = \bigcup_{i=0}^{3} \{P_i.get, F_i.use_l, F_{(i+1)\%4}.use_r\}$ and $\Gamma_{rel} = \bigcup_{i=0}^{3} \{P_i.rel, F_i.free_l, F_{(i+1)\%4}.free_r\}$.

Notice that several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behavior. One can add priorities to reduce non-determinism. In this case, one of the interactions with the highest priority is chosen non-deterministically.

Definition 3 (Priority). *Let $C = (Q, \Gamma, \rightarrow)$ be the behavior of the composite component $\Gamma(\{B_1, \ldots, B_n\})$. A* priority model $\prec$ *is a strict partial order on the set of interactions Γ. Given a priority model $\prec$, we abbreviate $(a, a') \in \prec$ by $a \prec a'$. Adding the priority model $\prec$ over $\Gamma(\{B_1, \ldots, B_n\})$ defines a new composite component $B = \prec\big(\Gamma(\{B_1, \ldots, B_n\})\big)$ noted $\texttt{prio}(C)$ and whose behavior is defined by $(Q, \Gamma, \rightarrow_\prec)$, where $\rightarrow_\prec$ is the least set of transitions satisfying the following rule:*

$$\frac{\boldsymbol{q} \xrightarrow{a} \boldsymbol{q'} \qquad \neg\big(\exists a' \in \Gamma, \exists \boldsymbol{q''} \in Q : a \prec a' \wedge \boldsymbol{q} \xrightarrow{a'} \boldsymbol{q''}\big)}{\boldsymbol{q} \xrightarrow{a}_\prec \boldsymbol{q'}}$$

An interaction a is enabled in $\texttt{prio}(C)$ whenever a is enabled in C and a is maximal according to $\texttt{prio}$ among the active interactions in C.

BIP provides both centralized and distributed implementations. In the centralized implementation, a centralized engine guarantees to execute only one interaction at a time, and thus conforms to the operational semantics of the BIP. The main loop of the BIP engine consists of the following steps: (1) Each atomic component sends to the engine its current location; (2) The engine enumerates the list of interactions in the system, selects the enabled ones based on the current location of the atomic components and eliminates the ones with low priority; (3) The engine non-deterministically selects an interaction out of the enabled interactions; (4) Finally, the engine notifies the corresponding components and schedules their transitions for execution.

Alternatively, BIP allows the generation of distributed implementations [3] where non-conflicting interactions can be simultaneously executed.

Definition 4 (BIP system). *A BIP system is a tuple $(B, \boldsymbol{q_0})$, where $\boldsymbol{q_0}$ is the initial state with $\boldsymbol{q_0} \in \bigotimes_{i=1}^{n} Q_i$ being the tuple of initial states of atomic components.*

For the rest of the paper, we fix an arbitrary BIP-system $(B, \boldsymbol{q_0})$, where $B = \prec\big(\Gamma(\{B_1, \ldots, B_n\})\big)$ with semantics $C = (Q, \Gamma, \rightarrow)$.

We abstract the execution of a BIP system as a trace.

Definition 5 (BIP trace). *A BIP trace $\rho = (\boldsymbol{q_0} \cdot a_0 \cdot \boldsymbol{q_1} \cdot a_1 \cdots \boldsymbol{q_{i-1}} \cdot a_{i-1} \cdot \boldsymbol{q_i})$ is an alternating sequence of states of Q and interactions in Γ; where $\boldsymbol{q_k} \xrightarrow{a_k} \boldsymbol{q_{k+1}} \in \rightarrow$, for $k \in [0, i-1]$.*

Given a trace $\rho = (\boldsymbol{q_0} \cdot a_0 \cdot \boldsymbol{q_1} \cdot a_1 \cdots \boldsymbol{q_{i-1}} \cdot a_{i-1} \cdot \boldsymbol{q_i})$, $\rho^{\boldsymbol{q_i}}$ (resp. ρ_i^a) denotes the i^{th} state (resp. interaction) of the trace, i.e., $\boldsymbol{q_i}$ (resp. a_i). Also, $\boldsymbol{\rho}(C)$ denotes the set of all the traces of an LTS C.

4 Problem Definition & Methodology

We frame the problem of run time enforcement as a sequential decision making (SDM) one, by which the BIP engine has to be guided to select the set of

interactions over extended execution traces that maximize a cumulative return. We formalize SDMs as the following five-tuple $\left\langle Q, \tilde{Q}, \Gamma, \rightarrow, R_+, R_-, \gamma \right\rangle$. Here, Q represents the set of all possible states, $\tilde{Q} \subseteq Q$ the set of "bad" states that need to be avoided, Γ the set of allowed interactions, and $\rightarrow$ represents the transition model. R_+ and R_- are two positive and negative scalar parameters, which allow us to define the reward function quantifying the selection of the engine. Clearly, the engine gets rewarded when in a state $\boldsymbol{q} \notin \tilde{Q}$, while penalized if $\boldsymbol{q} \in \tilde{Q}$. Using this intuition, one can define a reward function of the states written as:

$$\mathcal{R}(\boldsymbol{q}) = \begin{cases} R_+ & : \boldsymbol{q} \notin \tilde{Q} \\ R_- & : \boldsymbol{q} \in \tilde{Q}. \end{cases}$$

Given the above reward definition, we finally introduce $\gamma \in [0, 1)$ to denote the discount factor specifying the degree to which rewards are discounted over time as the engine interacts with each of the components.

At each time step t, the engine observes a state $\boldsymbol{q_t} \in Q$ and must choose an interaction $a_t \in \mathcal{A}_{\boldsymbol{q_t}} \subseteq \Gamma$, transitioning it to a new state $\boldsymbol{q_t} \xrightarrow{a_t} \boldsymbol{q_{t+1}}$ as given by $\rightarrow$ and yielding a reward $\mathcal{R}(\boldsymbol{q_{t+1}})$, where $\mathcal{A}_{q_t}$ denotes all enabled interactions from state q_t, i.e., $\mathcal{A}_{\boldsymbol{q_t}} = \{a \mid \exists \boldsymbol{q'} : \boldsymbol{q_t} \xrightarrow{a} \boldsymbol{q'} \in \rightarrow\}$. We filter the choice of the allowed interactions, i.e., $\mathcal{A}_{\boldsymbol{q_t}}$, at each time-step by an interaction-selection rule, which we refer to as the policy π. We extend the sequential decision making literature by defining policies that map between the set of states, Q, and any combination of the allowed interactions, i.e., $\pi : Q \rightarrow 2^{\Gamma}$, where for all $\boldsymbol{q} \in Q : \pi(\boldsymbol{q}) \subseteq \mathcal{A}_{\boldsymbol{q}}$. Consequently, the new behavior of the composite component, guided by the policy π, is defined by $C_\pi = (Q, \Gamma, \rightarrow_\pi)$, where $\rightarrow_\pi$ is the least set of transitions satisfying the following rule:

$$\frac{\boldsymbol{q} \xrightarrow{a} \boldsymbol{q'} \qquad a \in \pi(\boldsymbol{q})}{\boldsymbol{q} \xrightarrow{a}_{\pi} \boldsymbol{q'}}$$

The goal now is to find an optimal policy $\pi^\star$ that maximizes the *expected* total sum of the rewards it receives in the long run, while starting from an initial state $\boldsymbol{q_0} \in Q$. We will evaluate the performance of a policy π by: $\texttt{eval}(\pi|\boldsymbol{q_0}) = \mathbb{E}_{\boldsymbol{\rho}(C_\pi)}\left[\sum_{t=0}^{T} \gamma^t \mathcal{R}(\boldsymbol{q_{t+1}})\right]$, where $\mathbb{E}_{\boldsymbol{\rho}(C_\pi)}$ denotes the expectation under all the sets of all the allowed (by the policy π) possible traces, and T is the length of the trace. Notice that we index the value of the state by the policy π to explicitly reflect the dependency of the value on the policy being followed from a state $\boldsymbol{q_t}$. Interestingly, the definition of the evaluator asserts that the value of a state $\boldsymbol{q_t}$ is the expected instantaneous reward plus the expected discounted value of the next state. Clearly, we are interested in determining the optimal policy $\pi^\star$, which upon its usage yields maximized values for any $\boldsymbol{q_t} \in Q$. As such our goal is to determine a policy π that solves: $\pi^\star \equiv \max_\pi \texttt{eval}(\pi|\boldsymbol{q_0})$.

Finally, being in a state $\boldsymbol{q}$, we quantify the performance of the state-interaction pairs using the function $\mathbb{P} : Q \times \Gamma \rightarrow \mathbb{R}$. Given such a performance measure $\mathbb{P}$, the engine can follow the policy π, defined as follows:

$$\pi(\boldsymbol{q}) = \arg\max_{a}\{\mathbb{P}(\boldsymbol{q}, a) \mid a \in \mathcal{A}(\boldsymbol{q})\}. \tag{1}$$

In other words, given a state $\boldsymbol{q}$, policy π selects the enabled interaction that has maximum evaluation from that state. Clearly, an interaction must have a maximum evaluation when it is guaranteed that its execution will not lead to a bad state.

In what comes next, we define two methods capable of computing such performance measures, i.e., $\mathbb{P}$, (consequently policies) in finite as well as infinite state-space.

4.1 Finite State-Space - Value Iteration

Due to the number of possible policies at each time step, it is a challenge to compute the value for all possible options. Instead, we propose the application of a dynamic programming algorithm known as value iteration, summarized in Algorithm 1 to find the optimal policy efficiently.

In essence, Algorithm 1 is iteratively updating the performance measures of all the state-interaction pairs (until either (1) we reach a predefined bound, i.e., `bound`; or (2) the values are within a predefined ϵ), by choosing these interactions that maximize the instantaneous rewards, as well as the future information encoded through $V(q')$. Contrary to state-space-exploration algorithms, our method remedies the need to construct the full labeled transition system as we only require the knowledge of the successor state from a given state-interaction pair with no regard to its reachability properties. Notice also that though line 4 in Algorithm 1 requires a loop over all states computational time can be highly reduced by following a sampling-based procedure, where fractions of the state-space are considered. Notice, however, the successfulness of the attained policy comes hand-in-hand with the fraction of the state space sampled. In other words, the higher the fraction, the closer to optimality is the policy and vice versa.

Algorithm 1 Value Iteration Finite State Space

1: **Input:** Initialization of $V(\boldsymbol{q})$ for all $\boldsymbol{q} \in Q$, precision parameter ϵ
2: $\texttt{error} = \epsilon + 1$
3: **while** $\texttt{k} < \texttt{bound} \wedge \texttt{error} \geq \epsilon$ **do**
4: **for each** $\boldsymbol{q} \in Q$ **do**
5: **for each** $a \in \mathcal{A}_{\boldsymbol{q}}$ **do**
6: $\texttt{tmp} = \mathcal{R}(\boldsymbol{q'}) + \gamma V(\boldsymbol{q'})$, where $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q'}$
7: $\texttt{error} = \max(\texttt{error}, |\texttt{tmp} - \mathbb{P}(\boldsymbol{q}, a)|)$
8: $\mathbb{P}(\boldsymbol{q}, a) = \texttt{tmp}$
9: **end for**
10: $V(\boldsymbol{q}) = \max_{a \in \mathcal{A}_{\boldsymbol{q}}} \mathbb{P}(\boldsymbol{q}, a)$
11: **end for**
12: $\texttt{k} = \texttt{k} + 1$
13: **end while**

4.2 Infinite State-Space - Deep Value Iteration

The methodology detailed so-far suffers when considering infinite state-spaces as it requires exact representation of performance measures and policies. In general, an exact representation can only be achieved by storing distinct estimates of the return for every state-interaction pair. When states are continuous (i.e., components with variables or large state-space), such exact representations are no longer possible and performance measures need to be represented approximately.

Approximation in the continuous setting is not only a problem of representation. Two additional types of approximation are needed. Firstly, sample-based approximation is necessary in any of these frameworks. Secondly, the algorithm must repeatedly solve a difficult minimization problem. To clarify, consider Algorithm 1, where every iteration necessitates a loop over *every state-interaction pair*. When state space contains an infinite number of elements, it is impossible to loop over all pairs in finite time. Instead, a sample-based update that only considers a finite number of such pairs has to be used.

In this section, our goal is to develop an algorithm capable of avoiding the problems above. This ultimately leads us to a method for run-time enforcement operating in continuous state spaces. To commence, we introduce a function approximator, encoded through a neural network (NN), to represent a good approximation of performance measures of all state-interaction pairs. The goal of this approximator is to *autonomously generalize* over the state-space, such that similarly behaving states cluster together. Before commencing with our algorithm, we next introduce a concise introduction to NNs, accompanied with its generalization to the deep setting.

4.3 Neural Networks (NNs)

In an artificial NN, a neuron is a logistic unit, which is fed inputs through input wires. This unit can perform computations resulting in outputs that are transmitted through the output wires.

An artificial NN is simply a set of these logistic units strung together as shown in Figure 2. Each two layers are connected together using weight parameters. As such, the NN in Figure 2 possesses two weighting matrices, $\boldsymbol{\Theta}^{(1)}$ and $\boldsymbol{\Theta}^{(2)}$. Here, we used $\boldsymbol{\Theta}^{(l)}$ to denote the weights connecting layers l and $l+1$. Definitely, the dimensionality of $\boldsymbol{\Theta}^{(l)}$ depends on the number of units in each of the two layers[1].

For example, in our case, the dimension of the input layer is equal to the number of components (each input receives the current state of a component), and the dimension of the output layer is equal to the number of interactions. The number of hidden layers and the number of neurons per hidden layer can be configured depending on the functions to be learnt.

[1] In practice, a bias term is added to increase the expressiveness of the functions learnt by the NN.

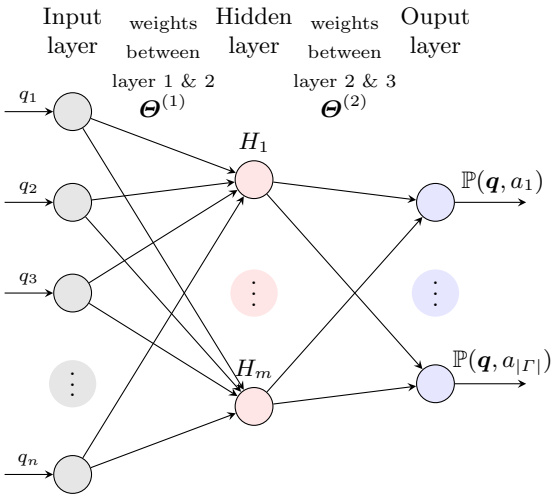


Fig. 2. A high-level depiction of an artificial NN.

Feed Forward. Given the notation introduced above, we are now ready to discuss the computations that are performed by a NN. Intuitively, between every two layers the inputs from the previous layer are, first, linearly (through the weight matrices) propagated forward and then nonlinearly transformed (through the sigmoids) to produce an output on the successor layer. Recursing this process, which we refer to as forward propagation, over the total layers of the network will produce an output on the final layer L.

Training & Backward Propagation. Having described feed forward propagation, the next step is to detail the strategy by which NNs determine the model parameters (i.e., the $\boldsymbol{\Theta}^{(l)}$ matrices – denoted by $\boldsymbol{\Theta}$, i.e., $\boldsymbol{\Theta} = \{\boldsymbol{\Theta^{(1)}}, \ldots, \boldsymbol{\Theta^{(L)}}\}$). In standard regression or classification problems, back-propagation is the algorithm adopted. Given an input data point, back-propagation commences as follows. First, forward propagation is executed and the network is made to output a value. This value is then compared to the real output from the data set producing an error. This error is then propagated backwards to every other layer and used to update connecting weights. Such updates typically involve gradient-based methods (e.g., stochastic gradients).

Unfortunately, the direct application of NNs in our context is challenging since the performance measures $\mathbb{P}$ has to build, through sampling, a *labeled* data set (with states being inputs and state-interaction values as outputs) to train on. The goal now is to determine at compile-time a good approximation of $\mathbb{P}$ through exploring an infinite state-space.

4.4 Deep Value Iteration – Infinite State Space

In Algorithm 2, we present a solution for approximating the performance measures $\mathbb{P}_{\boldsymbol{\Theta}}$[2] of all the state-interaction pairs in case of infinite state-space.

Algorithm 2 Deep-Value Iteration Infinite State Space

1: Initialize replay memory $\mathcal{D}$ to capacity N, and the NN weights randomly, K
2: **for** episode $= 1$ to M **do**
3: Set initial state to $\boldsymbol{q_0}$
4: **for** $t = 1$ to T **do**
5: With some probability ϵ select a random interaction a_t
6: With a probability $1 - \epsilon$ select interaction $a_t = \arg\max_{a \in \mathcal{A}_{\boldsymbol{q_t}}} \mathbb{P}_{\boldsymbol{\Theta}}(\boldsymbol{q_t}, a)$
7: Execute interaction a_t and observe reward, r_{t+1}, and successor state $\boldsymbol{q_{t+1}}$
8: Store transition $(\boldsymbol{q_t}, a_t, r_{t+1}, \boldsymbol{q_{t+1}})$ on replay memory $\mathcal{D}$
9: Sample random minibatch of transitions $(\boldsymbol{q_j}, a_j, r_{j+1}, \boldsymbol{q_{j+1}})$ of size N_2 from $\mathcal{D}$ and create output label by

$$y_j = \begin{cases} r_{j+1} & \text{if } \boldsymbol{q_{j+1}} \text{ is a bad state} \\ r_{j+1} + \gamma \max_{a \in \mathcal{A}_{\boldsymbol{q_{j+1}}}} \mathbb{P}_{\boldsymbol{\Theta}^-}(\boldsymbol{q_{j+1}}, a) & \text{if } \boldsymbol{q_{j+1}} \text{ is a correct state} \end{cases}$$

10: **end for**
11: Retrain network on the N_2 data points with y_j being the labels.
12: Update $\boldsymbol{\Theta^-}$ to $\boldsymbol{\Theta}$ every K episodes.
13: **end for**

In particular, we use an NN that takes a BIP state as input, encoded as a vector of size n, where n is the number of atomic components (i.e., the i^{th} input encodes the local state of atomic component B_i). The output of the NN encodes the performance measures $\mathbb{P}$ for each interaction. As such, the i^{th} output of the NN encodes the safety of executing interaction a_i.

On a high-level, the algorithm operates in two loops. The first is episode-based, while the second runs for a horizon T. At each episode, the goal is to collect relevant labeled data, encoding a trace of the system, to improve the approximation – encoded through the NN – of the performance measure, as summarized in lines 5-10 in the algorithm.

A trace is selected as follows. First, the algorithm selects an allowed interaction either randomly (line 5) with a probability ϵ (i.e., exploration) or by exploiting (line 6) the current estimate of the performance measure. In other words, given the current state $\boldsymbol{q}$, forward propagation is first executed to produce $\mathbb{P}_{\boldsymbol{\Theta}}(\boldsymbol{q}, a_i)$. As such, the enabled interaction that has a maximum performance measure is selected, i.e., $\arg\max_{a \in \mathcal{A}_{\boldsymbol{q}}} \mathbb{P}_{\boldsymbol{\Theta}}(\boldsymbol{q}, a)$. Next, the engine executes the interaction and stores both the dynamical transition and its usefulness (i.e., reward) in a replay memory data set, $\mathcal{D}$. We use a technique known as experience replay [12] where we store the transitions executed at each step,

[2] Note that $\mathbb{P}$ is indexed by $\boldsymbol{\Theta}$ as its output depends on $\boldsymbol{\Theta}$.

$(\boldsymbol{q_t}, a_t, r_{t+1}, \boldsymbol{q_{t+1}})$ in a replay memory $\mathcal{D}$. By using memory replay the behavior distribution is averaged over many of its previous transitions, smoothing out learning and avoiding oscillations or divergence in the parameters.

To ensure that the learning algorithm takes learning memory into account, we sample a set of size N_2, with the help of alternative NN (with weight matrices $\boldsymbol{\Theta}^-$). This is due to the fact that the backward propagation (used for training) discussed previously operates successfully for relatively "shallow" networks, i.e., networks with low number of hidden layers. As the number of these layers increases (i.e., deep NN), propagating gradients backward becomes increasingly challenging leading to convergence to local minima. To circumvent the above problems, we adapt a solution by which gradient updates are not performed at each iteration of the training algorithm. In particular, we assume additional knowledge modelled via an alternative NN that encodes previously experienced traces. This NN is used as a reference that we update after a preset number of iterations. As such, old knowledge encountered by the agent is not hindered by novel observations.

Consequently, we form a data set $\mathcal{D}$ (line 8) in preparation to retrain the original NN, while taking the history of traces into account. The process by which we generate these labels is in essence similar to finite value iterator. The main difference, however, is the usage of sample-based transitions to train a NN.

4.5 Fairness

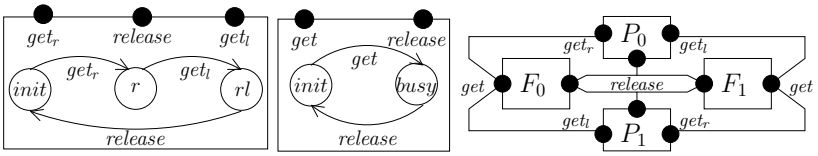
Deep value iteration allows to compute $\boldsymbol{\Theta}$, and hence $\mathbb{P}_{\boldsymbol{\Theta}}$ for all state-interaction pairs. As defined in Equation 1, the policy then can be defined using $\mathbb{P}$. For this, as we are dealing with real numbers, the same trace would be selected all the time by engine, which is running that policy. As such, other correct traces will not be reachable in the obtained system. For instance, given a global state, a policy would select the interaction leading to a state with maximum performance measure value, even though there exist other interactions leading to other correct traces. To remedy this, we define a fair policy that is allowed to deviate from the optimal policy with a degree of fairness. The fair policy is defined as follows.

$$\pi(\boldsymbol{q}) = \{a \mid a \in \mathcal{A}_{\boldsymbol{q}} \wedge \mathbb{P}_{\boldsymbol{\Theta}}(\boldsymbol{q}, a) \geq \mathtt{max_q} - \mathtt{fair}\}, \tag{2}$$

where, $\mathtt{max_q} = \max_{a \in \mathcal{A}_{\boldsymbol{q}}} \mathbb{P}_{\boldsymbol{\Theta}}(\boldsymbol{q}, a)$. `fair` is the tolerance against the optimal policy. The value of `fair` depends on (1) the value of good and bad rewards, and (2) the horizon used in deep value iteration algorithm. Clearly, the more fairness the more deviation from the optimal policy we get.

5 Experimental results

In this section, we present `RERL` an implementation of our proposed approach and its evaluation in terms of (1) accuracy of avoiding bad states and, (2) compilation and runtime efficiency.



(a) Philosopher (b) Fork (c) Dining philosophers in BIP

Fig. 3. Dining philosophers with possible deadlock.

5.1 Implementation

`RERL` is an implementation (developed in Java) of our method with several modules. The implementation is available at http://staff.aub.edu.lb/~mj54/rerl. `RERL` is equipped with a command line interface that accepts a set of configuration options. It takes the name of the input BIP file, a file containing the bad states (explicitly or symbolically) to be avoided, and optional flags (e.g., discount factor, number of episodes, horizon length), and it automatically generates a C++ implementation of the input BIP system embedded with a policy to avoid bad states.

```
> java -jar RERL.jar [options] input.bip output.cpp badStates.txt
```

5.2 Evaluation

Dining philosophers. Figure 3 shows an example of dining philosophers modeled in BIP that may deadlock. A deadlock will arise when philosopher P_0 allocates its right fork F_0, then philosopher P_1 allocates its right fork F_1. That is, a deadlock state is defined as the concatenation when all the philosophers allocate their right forks. We used `RERL` to enforce deadlock freedom on this example. We vary the number of philosophers from 2 to 47 increasing by 5. The total number of states is equal to 6^n, where n is the number of philosophers. Clearly, value iteration method would explode when the number of philosophers is greater than 12. As for the deep value iteration, we ran it with 50 `epochs`[3], 50 hidden neuron[4] units and 5 as degree of fairness. The degree of fairness was chosen to be consistent with the good (+1) and bad reward (−1). Then, we run the system until it executes 10^5 iterations. The normal implementation always enters the deadlock state whereas when we apply deep value iteration the obtained implementation always avoid the deadlock state. We also evaluated the runtime overhead induced by the policy, which at every computation step needs to do a forward propagation on the trained neural network to select the best next interactions. Moreover, we compare this overhead against RE-BIP, a tool-set to enforce properties in BIP systems, but which is limited to only a one-step recovery. In this example, one-step recovery is sufficient to avoid deadlock, however, as we will see later it fails

[3] One epoch consists of one full training cycle on the training set.
[4] We use fine-tuning technique to select these parameters.

in the second benchmark as a k step recovery is needed. Table 1 summarizes the runtime execution times in case of infinite (while varying number of hidden neuron units), i.e., deep value iteration, finite, i.e., value iteration, and RE-BIP. Clearly, the infinite-based implementation drastically outperforms RE-BIP, when the number of hidden neuron units is less than 100. Although the finite-based outperforms RE-BIP and guarantees to enforce correct execution, it fails when the size of the system becomes large.

Table 1. Execution times (in seconds).

Nb of Philo.	**Infinite**					**Finite**	**RE-BIP**
	10	50	100	500	1000		
2	0.7	2.8	5.5	27	55	1.2	29
7	1.4	6	11.3	58	130	12.2	72
12	2.3	9	17.5	90	186	43	122
17	3	11.8	23.2	121	251	NA	173
22	3.9	15.6	29.3	154	322	NA	269
27	5.5	18.7	35	190	405	NA	301
32	5.9	22.5	41.8	229	491	NA	407
37	7.1	25	48.5	279	567	NA	450
42	7.7	28.2	54.9	325	648	NA	566
47	9.7	32.6	60.5	396	764	NA	652

Pair-wise synchronized robots. In this benchmark, we model a set of robots placed on a map of size $n \times n$. Initially, all robots are placed at position $(0, 0)$. We consider that robots can only move up or to the right, that is, cannot go back after taking a move. Moreover, robot i must synchronize with either robot

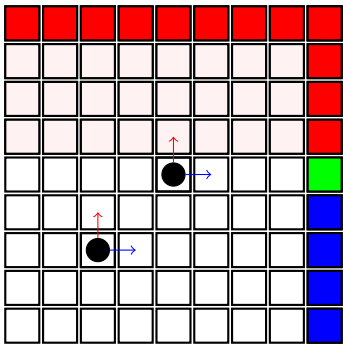


Fig. 4. Map of pair-wise synchronizing robots

$i - 1$ or $i + 1$ (modulo the number of robots) in order to move and both must

Table 2. Evaluation of two robots with different grid sizes.

Grid Size	Infinite		Finite		Standard
	Succ. %	Conv. (s)	Succ. %	Conv. (it.)	Succ. %
5	96.8	0.85	100	7	61.7
9	98.6	0.98	100	9	63.3
13	98.5	1.09	100	26	61.6
17	99.4	1.13	100	34	58.4
21	99.3	1.29	100	42	63.2
25	99.9	1.4	100	50	59.2
29	99.6	1.5	100	58	61.8

move with the same direction. Clearly, this would require to create controllers to allow robots to follow that specification. The state of each robot can be modeled by two integers denoting the current position of the robot. We also assume the grid has mines (i.e., bad states) at the top row and the top half most left of the map (i.e., red places in Figure 4). Bottom half most left places are considered safe. Also, we assume that the green location has an exit, which allows the robot to safely exit the map.

We have modeled robots and their pair-wise synchronization using BIP by allowing them to move to any border locations. Then, we define bad states (i.e., red location) and the goal is to generate a policy that allows robots not go to a bad state. Notice that RE-BIP cannot be used to avoid states as if a robot enters a location on the top half of the map, then 1-step recovery would try all the possible next actions and then fail. For instance, the robot (black circle) on the top has two choices, either moves right or up. The two choices lead the robot to go to a correct state. However, if the robot would take the move up action, it will enter a region where 1-step recovery will fail. We have tested `RERL` using value iteration and deep value iteration by varying the size of the map and the number of robots. Tables 2, 3, 4 depict (1) the success rate when using deep value iteration, value iteration and standard implementation, (2) the time needed to converge in case of deep value iteration, and (3) the number of iterations needed to converge in case of finite value iteration. We ran each configuration 10000 times to compute the success rate. We notice that the value iteration provides a success rate of 100%, however, it fails when the size of the system increases. As for the deep value iteration, the system is learning to avoid bad states or states that could lead to bad states and it achieves a high success rate. For instance, if we take a map with 29×29 grid size and 8 robots (i.e., 841^8 possible states), the standard implementation has 15.1% success rate whereas when using deep value iteration we reach 95.6% success rate. As the state space in this example has a well-defined structure, we only needed 10 hidden neuron units to train our network by using deep value iteration algorithm. For this, we notice the efficiency of the compile time, e.g., only 3.6 seconds are needed to train a system consisting of 841^8 states and to reach a 95.6% success rate.

Table 3. Evaluation of four robots with different grid sizes.

Grid Size	Infinite		Finite		Standard
	Succ. %	Conv. (s)	Succ. %	Conv. (it.)	Succ. %
5	95.5	0.9	100	7	30.5
9	93.9	1.08	NA	NA	30.7
13	94.7	1.23	NA	NA	30.6
17	97.8	1.52	NA	NA	28.6
21	97.9	1.82	NA	NA	29.8
25	98.1	2.2	NA	NA	30.1
29	97.8	2.5	NA	NA	30.1

Table 4. Evaluation of eight robots with different grid sizes.

Grid Size	Infinite		Finite		Standard
	Succ. %	Conv. (s)	Succ. %	Conv. (it.)	Succ. %
5	92.9	1.1	NA	NA	12.2
9	93.9	1.5	NA	NA	13.8
13	97.4	1.6	NA	NA	15.1
17	95.3	2.2	NA	NA	13.8
21	94.9	3.0	NA	NA	17.1
25	94.8	3.3	NA	NA	14.2
29	95.6	3.6	NA	NA	15.1

6 Conclusions and perspectives

In this paper, we introduced a new technique that combines static analysis and dynamic analysis with the help of machine learning techniques, in order to optimally ensure the correct execution of software systems. Experimental results show that it is possible to learn reachability of bad behaviors by only exploring part of the system. For future work, we consider several directions. First, we plan to study more expressive properties (e.g., liveness). Second, we consider to generate a partial state semantics, and hence, allow to automatically generate multi-threaded implementations. Third, we consider to generate decentralized policies to facilitate the generation of efficient distributed implementations.

References

1. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
2. Bensalem, S., Bozga, M., Nguyen, T., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Computer Aided Verification, 21st International Conference, CAV Grenoble, France. Proceedings. pp. 614–619 (2009)
3. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing 25(5), 383–409 (2012)

4. Charafeddine, H., El-Harake, K., Falcone, Y., Jaber, M.: Runtime enforcement for component-based systems. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 1789–1796 (2015)
5. Clarke, E.M.: My 27-year quest to overcome the state explosion problem. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA. p. 3 (2009)
6. Clarke, E.M., Klieber, W., Novácek, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification, LASER, International Summer School, Elba Island, Italy, Revised Tutorial. pp. 1–30 (2011)
7. Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Software and System Modeling 14(1), 173–199 (2015)
8. Falcone, Y., Zuck, L.D.: Runtime verification: the application perspective. STTT 17(2), 121–123 (2015)
9. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Conference Record of POPL 2002: The 29th Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 191–202 (2002)
10. Huang, C., Peled, D.A., Schewe, S., Wang, F.: A game-theoretic foundation for the maximum software resilience against dense errors. IEEE Trans. Software Eng. 42(7), 605–622 (2016)
11. Katz, G., Peled, D.A.: Synthesizing, correcting and improving code, using model checking-based genetic programming. In: Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings. pp. 246–261 (2013)
12. Lin, L.J.: Reinforcement Learning for Robots Using Neural Networks. Ph.D. thesis, Pittsburgh, PA, USA (1992), uMI Order No. GAX93-22750
13. Peled, D.: Automatic synthesis of code using genetic programming. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I. pp. 182–187 (2016)
14. Peled, D.: Using genetic programming for software reliability. In: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. pp. 116–131 (2016)
15. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016. pp. 1628–1633 (2016)
16. Pinisetty, S., Tripakis, S.: Compositional runtime enforcement. In: NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings. pp. 82–99 (2016)
17. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989. pp. 179–190 (1989)
18. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II. pp. 746–757 (1990)
19. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings. pp. 337–351 (1982)