RAVEN: Reinforcement Learning for Generating Verifiable Run-Time Requirement Enforcers for MPSoCs

Khalil Esper[1]
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Jan Spieck[1]
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Pierre-Louis Sixdenier
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

In embedded systems, applications frequently have to meet non-functional requirements regarding, e.g., real-time or energy consumption constraints, when executing on a given MPSoC target platform.

Feedback-based controllers have been proposed that react to transient environmental factors by adapting the DVFS settings or degree of parallelism following some predefined control strategy. However, it is, in general, not possible to give formal guarantees for the obtained controllers to satisfy a given set of non-functional requirements. Run-time requirement enforcement has emerged as a field of research for the enforcement of non-functional requirements at run-time, allowing to define and formally verify properties on respective control strategies specified by automata. However, techniques for the automatic generation of such controllers have not yet been established.

In this paper, we propose a technique using reinforcement learning to automatically generate verifiable feedback-based enforcers. For that, we train a control policy based on a representative input sequence at design time. The learned control strategy is then transformed into a verifiable enforcement automaton which constitutes our run-time control model that can handle unseen input data. As a case study, we apply the approach to generate controllers that are able to increase the probability of satisfying a given set of requirement verification goals compared to multiple state-of-the-art approaches, as can be verified by model checkers.

2012 ACM Subject Classification Computer systems organization → Multicore architectures; Theory of computation → Linear logic; Theory of computation → Modal and temporal logics; Hardware → Finite state machines; Computer systems organization → Self-organizing autonomic computing; Theory of computation → Verification by model checking; Mathematics of computing → Probabilistic representations; Computing methodologies → Reinforcement learning

Keywords and phrases Verification, Runtime Requirement Enforcement, Reinforcement Learning

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.7

Funding This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research-Foundation) – Project Number 146371743 - TRR 89 Invasive Computing.

[1] Khalil Esper and Jan Spieck both contributed equally to this work.

© Khalil Esper, Jan Spieck, Pierre-Louis Sixdenier, Stefan Wildermann, and Jürgen Teich;
licensed under Creative Commons License CC-BY 4.0

Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023).
Editors: Federico Terraneo and Daniele Cattaneo; Article No. 7; pp. 7:1–7:16

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Current multi-processor on-chip (MPSoC) platforms offer abundant computational and storage resources that necessitate new programming paradigms such as invasive computing [39, 1] for isolating applications to handle architectural interferences between applications. Hybrid mapping approaches [35, 33, 29] have emerged for mapping of applications onto multi-core systems in the presence of uncertainty [34, 36].

Despite inter-application resource isolation schemes, applications are still exposed to variances in the system state (e.g., due to scheduler or caching effects) [6]. According to [38], another source of uncertainty is the varying workload induced by the input data (e.g., different workloads for different image inputs). As an example, throughput jitter in virtual and augmented reality applications may not only be an annoying user experience, but even cause dizziness or a headache to a user.

Run-time Requirement Enforcement (RRE) [40] is a field of research with the aim to control the non-functional properties of execution of a program within desired bounds. Such techniques dynamically steer control knobs, e.g., voltage/frequency settings, in reaction to observed changes in the system state to keep the non-functional properties of execution within the desired range. RRE allows a user to specify bounds on execution properties of an application on a multi-core platform using so-called *requirements* [38], i.e., expressions on non-functional properties such as desired corridors on latency or energy consumption. Recently, techniques have been proposed in [9, 10, 11] to formally verify the satisfaction or violation of non-functional requirements of RRE techniques at design time. In order to apply formal methods such as model checking, *finite state machines (FSMs)* are used to formally specify control strategies. However, techniques for automatically generating FSMs for RRE that either always guarantee the satisfaction of a set of given non-functional requirements in case of *strict enforcement* or at least guarantee a certain probability of satisfying executions in case of *loose enforcement* [40] have not yet been established.

In this realm, different machine-learning-based techniques have been proposed for the dynamic control of program executions [25, 26]. In an offline phase, a controller behaviour is learned to optimize a set of given non-functional objectives that can be used at run-time to control the application. However, the above approach cannot provide any formal guarantees regarding the ability or strictness to fulfill a set of given requirements, i.e., constraints on non-functional execution properties. In this paper, we propose a technique using reinforcement learning to generate FSMs for RRE with formally verifiable guarantees by training and optimizing an FSM controller to be generated based on input sequences at design time. Based on a formal characterization of input variation at run time, the generated FSM controllers for RRE can then also be formally verified at design time.

Contributions: The main contributions of this paper can be summarized as follows:

1. Using reinforcement learning for FSM-based RRE generation: Based on training sequences, reinforcement learning is used to adapt an initial FSM model towards satisfaction (or improvement of the satisfaction probability) of a given set of non-functional program execution properties formulated as verification goals.
2. During learning, the RRE strategy is regularly transformed into an FSM and formally verified according to the set of verification goals. The offline learning phase stops once all goals are satisfied. Alternatively, based on a user-defined exit condition.
3. In a case study, the approach to generate formally verified FSM-based RRE controllers is compared to state-of-the-art enforcer designs.

2 Fundamentals

In the following, important notions and definitions are introduced.

2.1 FSM-based RRE

Non-functional requirements should be satisfied during each program execution on a given MPSoC platform even when the environmental input is varied. According to [10], let the size of the input of a given program for each discrete execution k be given by an *environment feature vector* $i(k) \in \mathcal{I}$, where $\mathcal{I}$ is called the *environment space*. Moreover, assume that for preventing or as a countermeasure against violations of a set of non-functional requirements, an enforcer can vary the number n of cores allocated to execute an application program as well as the voltage/frequency setting m of these cores. We call such a setting (n, m) a configuration c and the set of available configurations available on a given MPSoC platform the *configuration space* C. Figure 1 shows the concept of feedback-based RRE according to [10] which serves as the base model also in this paper. Illustrated is a multi-core system stimulated by input from an environment and reacting to violation of a number of requirement using an enforcement FSM that determines the configuration $c(k+1)$ for the $(k+1)$th execution accordingly.

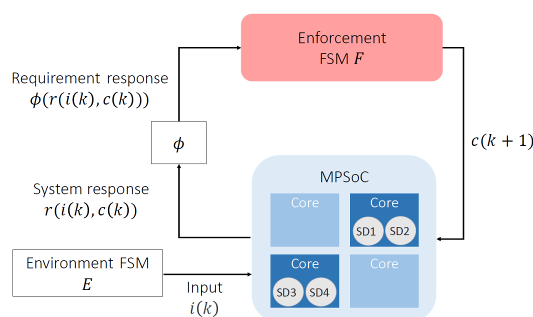

Figure 1 Illustration of feedback-based RRE. A system response vector r is mapped to a binary requirement response vector ϕ such that the enforcement FSM F controls the next configuration $c(k+1) \in C$. Adapted from [10].

2.1.1 Formal Definitions

Assume that the k-th execution of a program on an MPSoC yields H execution properties of interest (e.g., latency and energy consumption). These properties depend on the input data $i(k) \in \mathcal{I}$ and the system configuration $c(k) \in C$. For the purpose of RRE, the *system-under-control* can be abstracted by a single function called *system response function* $r : \mathcal{I} \times C \rightarrow \mathbb{R}^H$ (see [10]). Thus, the system response $r(i(k), c(k)) = (o_1(k), \ldots, o_H(k))$ at execution k is a vector of the H relevant execution properties (see Figure 1). According to [41], requirements can be specified for each property o_h, $h \in \{1, \ldots, H\}$, typically in terms of a lower bound LB_{o_h} and an upper bound UB_{o_h} that should not be violated. Such intervals can be described by two propositions φ_h^{LB} and φ_h^{UB} as follows:

$$\varphi_h^{LB}(o_h(k)) = (LB_{o_h} \leq o_h(k)) \quad (1)$$

$$\varphi_h^{UB}(o_h(k)) = (o_h(k) \leq UB_{o_h}) \quad (2)$$

In Equation (1) and Equation (2), LB_{o_h} and UB_{o_h} denote a user-given lower, respectively, upper bound on the execution property o_h. The information about which proposition is fulfilled and which is violated at the k-th execution can then be described by a binary vector denoted by *requirement response* ϕ (see Figure 1). It is obtained from the system response $r(i(k), c(k)) = (o_1(k), \ldots, o_H(k))$ using the *requirement response function* [10]:

$$\beta := \phi\,(o_1(k), \ldots, o_H(k)) = \big(\varphi^{LB}(o_1(k)), \varphi^{UB}(o_1(k)), \ldots,\\ \varphi^{LB}(o_H(k)), \varphi^{UB}(o_H(k))\big) \in \{0,1\}^{2H}. \tag{3}$$

This binary requirement response vector β specifies for each proposition to be satisfied for each execution k the input to the enforcement finite state machine (FSM) F, as illustrated in Figure 1. F reacts by computing the next configuration $c(k+1) \in C$ to enforce the desired non-functional properties for the next execution $k+1$. Formally, an enforcement FSM is defined as follows.

▶ **Definition 1** ([10]). *An* enforcement FSM (F) *is a deterministic finite state machine (Moore machine) that can be described by a 6-tuple $(Z, z_0, B, \delta, C, \gamma)$:*
- *Z is a finite set of states.*
- *$z_0 \in Z$ is the initial state.*
- *B is the input alphabet.*
- *δ is the transition relation: $\delta \subseteq B \times Z \times Z$ with $(\beta, z, z') \in \delta$ representing a transition from z to z' under input β.*
- *C is the output alphabet, also called configuration space.*
- *γ is the output function, which maps each state to the output alphabet: $\gamma : Z \to C$.*

Instead of verifying an enforcement strategy described by an enforcement automaton F for RRE just for individual input traces, the authors in [10] proposed rather to analyze families of traces. The input variation of the environment is modeled by a discrete-time Markov Chain called *environment FSM*, after partitioning the environment space of inputs $\mathcal{I}$ into a set P of disjoint partitions $p \in P$ with $p \subseteq \mathcal{I}$. The partitions are constructed such that all inputs $i \in p$ assigned to the same partition always deliver the same binary requirement response $\phi(r(i(k), c(k)))$ in each configuration $c \in C$. These partitions p then define a discrete state space of a discrete-time Markov chain E. Transitions between states reflect the probabilities of observable variations in environmental input from state to state. The environment FSM E can equally be seen as a generator of potential input traces that an RRE FSM F shall be evaluated for. But rather than evaluating a single or comparing multiple enforcer FSMs based on just individual sample traces, we want to argue first about quality of enforcers rather for all input traces that a system can potentially undergo. Second, rather than simulating such input traces to generate statistics, we propose to apply symbolic techniques, i.e., probabilistic model checking for our analysis.

2.1.2 Verification Goals

Verification goals (VGs) can then be specified to compare different enforcement strategies regarding their quality to satisfy the given set of requirements. VGs are formulated over the two propositions φ_h^{LB} and φ_h^{UB}, see Equation (1) and Equation (2), using temporal logic [5] or PCTL [2, 17]. Examples of such verification goals of interest (one is applied in case of strict enforcement, the subsequent ones for loose enforcement) are [10]:
- $AG(\varphi)$: φ should always hold.
- $AF(\varphi)$: φ should eventually hold.

- $\mathcal{P}_{=?}[\neg\varphi \;\rightarrow\; F^{\leq\lambda}(\varphi)]$ denoting the probability of returning to a requirement-satisfying configuration state (φ) from a violating one ($\neg\varphi$) in no more than λ steps, i.e., next executions.
- $\mathcal{P}_{=?}[G^{\leq\lambda}\,(\neg\varphi_L)]$ denoting the probability of λ consecutive violations of φ.
- $\mathcal{S}_{=?}[\neg\varphi]$ denotes the steady-state probability of violating φ.

2.2 Reinforcement Learning

Reinforcement Learning (RL) [37] is a Machine Learning paradigm dealing with how an *agent* shall act in an environment in order to maximize a cumulative reward. An agent is supposed to improve its ability to solve a problem (defined via a reward function) through trials-and-errors, similar to how humans and animals learn. At its core, RL models a problem as a *Markov Decision Process (MDP)*, representing the environment, which the agent interacts with and observes. At each time step, the environment resides in a state $\upsilon \in \Upsilon$, based on which the agent then selects an action $a \in A$ according to its internal policy π that will put the state in a successor state $\upsilon' \in \Upsilon$. The resulting sequence of states and actions performed in the environment $\tau = (\upsilon_0, a_0, \upsilon_1, a_1, ...)$ is called a *trajectory* τ. Some key components when performing RL training are:

1. A *policy* $\pi : \Upsilon \times A \rightarrow [0,1]$ that defines the behavior of an agent, i.e., the probability to take each available action $a \in A$ for each state $\upsilon \in \Upsilon$. A policy π can have parameters θ (then denoted as π_θ), be stochastic (e.g., the ϵ-greedy policy in Equation (4)) or deterministic.
2. A *reward signal* $\xi : \Upsilon \times A \rightarrow \mathbb{R}$ is the feedback sent by the environment to the agent when it takes an action a in a state υ, ergo it represents the immediate goal of the agent.
3. An *action-value function* $Q^\pi : \Upsilon \times A \rightarrow \mathbb{R}$, which predicts the cumulated reward that could be obtained on the long run if the agent takes decision a in current state υ when following policy π.
4. In some cases, a *model* of the environment that allows for predicting rewards $\xi(\upsilon, a)$.

The most investigated way of solving a RL problem is to accurately estimate the action-value function. While doing so, an agent has to explore the state space Υ to receive the rewards. To that regards, there is a trade-off to be made between *exploration*, i.e., taking a random action at a given state υ, and *exploitation*, i.e., performing the action a that will maximize the expected cumulative reward $Q^\pi(\upsilon, a) = \mathbb{E}_{\tau\sim\pi}[\sum_{(\upsilon_t,a_t)\in\tau} \xi(\upsilon_t, a_t) \mid \upsilon_0 = \upsilon, a_0 = a]$ following the trajectory τ sampled from the policy π starting from state $\upsilon_0 = \upsilon$ and action $a_0 = a$. Too much exploration will increase the training time (as it is not different from a random selection of actions), whereas too much exploitation might lead to convergence to a local optimum. As an example, a common policy used (notably in Q-Learning) to balance exploration and exploitation is called *ϵ-greedy policy* and is presented in Equation (4). Here, $a \in_R A$ describes the sampling of a random action from A based on the probability distribution R.

$$a = \begin{cases} \arg\max_{a\in A} Q(\upsilon, a) & \text{with probability } (1-\epsilon) \\ a \in_R A & \text{with probability } \epsilon \end{cases} \tag{4}$$

In this paper, we will consider as an example a widely used algorithm, *Q-Learning*. Q-Learning [43] is a model-free algorithm which learns the action-value function, i.e., the Q-function. A common implementation is to have a *Q-table* storing all the values of Q. The algorithm steps are described in Algorithm 1 (in the appendix). An episode *ep* comprises multiple iterations, in which actions are picked and taken until υ is a terminal state or a given

number of maximum iterations is performed. Observing the reaction of the environment yields the successor state υ' and reward $\xi(\upsilon, a)$. The Q-table is then updated according to Equation (5).

$$Q(\upsilon, a) \leftarrow Q(\upsilon, a) + \alpha \cdot \left(\xi(\upsilon, a) + \kappa \cdot \max_a Q(\upsilon', a) - Q(\upsilon, a)\right) \tag{5}$$

The *learning rate* $\alpha \in [0, 1]$ determines how much of the newly acquired information should replace the current knowledge. The *discount factor* $\kappa \in [0, 1]$ represents how much influence future rewards have on the current optimization step compared to the instant one. For example, a value of 0 will make the agent "short-sighted", while a value of 1 will make it aim for an endgame goal.

One limitation of Q-learning is its discrete nature, which, when used to solve continuous problems, either leads to a state explosion or suboptimal performance due to a down-sampling of both actions and states. One way to solve continuous problems using this algorithm is to use deep artificial neural networks as approximators for the Q-function. This is referred to as Deep Q-Learning [14].

3 Reinforcement Learning for the Generation of Run-Time Requirement Enforcers

The structure of our verifiable RRE generation approach and optimization is depicted in Fig. 2. During a training phase (see upper part of the figure), an enforcement agent learns an enforcement strategy aiming to satisfy a set of verification goals defined over a set of given requirements like latency, energy, or power. The training is based on an input data sequence $I = \{i(1), i(2), ...\}$ generated by a Markov chain. To assess the satisfaction of the verification goals, the policy of the enforcement agent is periodically transformed (i.e., after a specified number of training iterations n_{update}) into an enforcement FSM and the verification goals VG are subsequently formally verified using a model checker (see lower part of Fig. 2). Since verifying the enforcement goals can be a time-intensive task, we employ surrogate functions in executions k between such verification checks to estimate the verification goals VG between the actual model checks to speed up the training process. The training phase stops either when a run-time requirement enforcer with verified VGs has been found, which then can be deployed in the field (*run-time phase*). Else, a user can determine the termination of the learning phase.

3.1 Learning phase

Our enforcement agent is defined by a set of states Υ, a set of actions A, and finally a reward function ξ that assesses the quality of choosing an action in the current state (see Section 2.2). An action $a(k)$ of our agent is determined by the selection of the configuration $c(k+1) = (n, m)$ of number of cores n and power mode m to be applied for processing the next input data $i(k+1)$. The set of actions A is thereby defined equal to the configuration space C, i.e., $A = C$. A state $\upsilon \in \Upsilon = B \times C$ is given by a pair of configuration $c \in C$ and corresponding requirement response $\beta \in B$ that indicates which requirements were fulfilled after having processed a given input data i in configuration c and which not. As a consequence, the number of states is given by: $|\Upsilon| = |B| \cdot |C|$.

Since we do not require any functional execution properties of the related programs for training the enforcement agent, our approach can also handle black-box applications. The feedback about the satisfaction of a set of verification goals is encapsulated into a reward

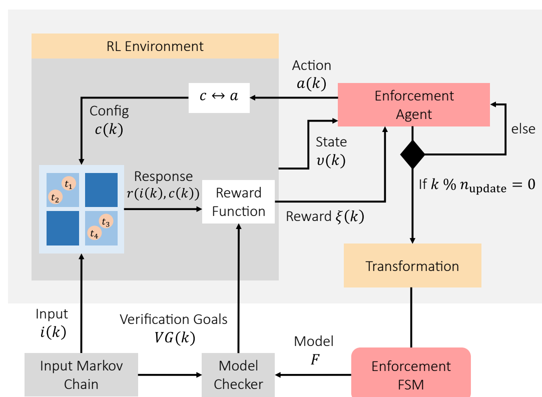

Figure 2 Enforcer generation based on Reinforcement Learning.

function ξ_η that associates a chosen action $a \in A$ with a quality assessment in the form of a numeric reward. In the following Equation 6, the reward is defined as a weighted sum of a *verified reward* ξ_{ver} – measured by transforming the enforcement agent into an enforcement FSM and employing a model checker – and a *surrogate reward* ξ_{sur} that provides a probabilistic estimation of the probabilities of fulfilling the verification goals based on the history of an already processed input data sequence. Consequently:

$$\xi_\eta(a(k)) = \eta \cdot \xi_{\text{sur}}(k) + (1 - \eta) \cdot \xi_{\text{ver}}(k). \tag{6}$$

With an increasing number of training iterations, the influence of the model checked reward ξ_{ver} should increase, as the surrogate reward merely serves as an estimate. This can be implemented by decaying η with increasing episode numbers ep, e.g., exponentially by $\eta = \eta_0 \cdot e^{-ep \cdot dec}$, with $\eta_0 \in [0, 1]$ being the initial value and $dec \in \mathbb{R}$ a decay hyperparameter.

Moreover, we define the verified reward ξ_{ver} as the weighted sum of given requirement verification goals VG of our model at iteration k and obtained by applying probabilistic model checking to an enforcer FSM obtained by a model transformation described in Section 3.2:

$$\xi_{\text{ver}}(k) = \sum_{\omega=1}^{|VG|} \varsigma_\omega \cdot VG_\omega(k). \tag{7}$$

In the case of a verification goal for strict enforcement, VG_ω represents a binary value indicating whether the verification goal was met $VG_\omega = 1$ or missed $VG_\omega = 0$, while in case of loose enforcement, $VG_\omega \in [0, 1]$ denotes a probability of meeting the verification goal. Furthermore, the weight ς_ω should be chosen negative when the associated verification goal VG_ω shall be minimized and positive when VG_ω shall be maximized.

In contrast to the verified reward ξ_{ver} that is updated periodically every n_{update} iterations, the surrogate reward ξ_{sur} is computed in each iteration k due to being an derivative of the function f_{est} at point k estimating the verification goals based on the history $\mathfrak{H} = (1, ..., k)$ of the input data i and agent trajectory τ up to the current action $a(k)$:

$$f_{\text{est}}(k) = \vartheta(k) \cdot \sum_{\omega=1}^{|VG|} \varsigma_\omega \cdot \mathbb{E}[VG_\omega \mid (i(y))_{y \in \mathfrak{H}}, (v_y, a_y)_{y \in \mathfrak{H}} \in \tau]. \tag{8}$$

With that, the surrogate reward $\xi_{\text{sur}}(k)$ is defined as:

$$\xi_{\text{sur}}(k) = \Delta f_{\text{est}}(k) = f_{\text{est}}(k) - f_{\text{est}}(k - 1). \tag{9}$$

Loose verification goals VG_ω can be estimated by using the empirical probability, i.e., associating the number of occurrences of verification goal violations over the history $\mathfrak{H}$. Since the accuracy of our probabilistic estimation increases with larger sizes of the regarded history $k = |\mathfrak{H}|$, we weight the surrogate function by a factor $\vartheta(k) \in [0, 1]$ that scales the reward with k:

$$\vartheta(k) = 1 - e^{-u \cdot k}, u \in \mathbb{R}^+. \tag{10}$$

Above function covers the range of $\vartheta(0) = 0$ up to $lim_{k\to\infty}\vartheta(k) = 1$ with $u \in \mathbb{R}^+$ steering the gradient of the scaling function. Strict verification goals are set to zero in case any violation happens in the history, and to one otherwise.

Example. Let the verification goal $VG_L := \mathcal{S}_{=?}[\varphi_L]$ be given, the goal being to minimize the probability of violating a latency requirement ϕ_L. This goal can be estimated as

$$\mathbb{E}[VG_L] = \frac{\sum_{y\in\mathfrak{H}} \xi_{\varphi_L}(k)}{|\mathfrak{H}|} \tag{11}$$

We can then define the reward for the requirement φ_L as:

$$\xi_{\varphi_L}(k) = \begin{cases} 1 & if\, \varphi^{UB}(o_L(i(k), a(k))) \wedge \varphi^{LB}(o_L(i(k), a(k))) \\ 0 & else \end{cases} \tag{12}$$

With this formalization of our reinforcement agent, we can choose a suitable reinforcement learning implementation from the literature for performing the training procedure. In case of a low cardinality of configurations $|C|$, simple approaches as Q-learning are viable. Else, more sophisticated model-free deep learning procedures as proximal policy optimization (PPO) [31] or soft actor-critic (SAC) [16] are recommended.

3.2 Transformation

This section describes how to transform a trained reinforcement learning agent into an enforcer FSM that can be formally verified. First, we need to transform our reinforcement learning agent states Υ into a set of enforcement FSM states Z. Second, we need to transform our agent policy into an FSM transition relation δ. Note that we can only transform reinforcement agents into a verifiable enforcer FSM for discrete action and state spaces.

We generate one unique enforcer FSM state $z_c \in Z$ of a Moore FSM per configuration $c \in C$, described by the bijective function $\zeta : C \leftrightarrow Z$. The FSM transition relation $\delta : B \times Z \to Z$ determines a next state from a current state based on the requirement response β. Since each state represents uniquely exactly one configuration, we can reformulate this relation as $\delta : B \times C \to C$, i.e., we have to determine for each configuration the best-suited subsequent configuration in dependence of the requirement response β. With the reinforcement states defined as $\Upsilon = B \times C$ and associated actions given by $A = C$, we propose to derive the state transitions by determining the best action per state (β, c) of our reinforcement learning agent. In particular, for each enforcer FSM state $z = \theta(c)$ corresponding to a configuration, we create one outgoing transition per reinforcement state $(\beta, c) \in \Gamma$ as follows:

$$\delta = \{(\beta, \zeta(c), \zeta(a)) \mid (\beta, c) \in \Upsilon \wedge a = \varrho(\beta, c))\}. \tag{13}$$

The best action per state $\varrho : \Upsilon \to A$ can be extracted from the trained agent policy.

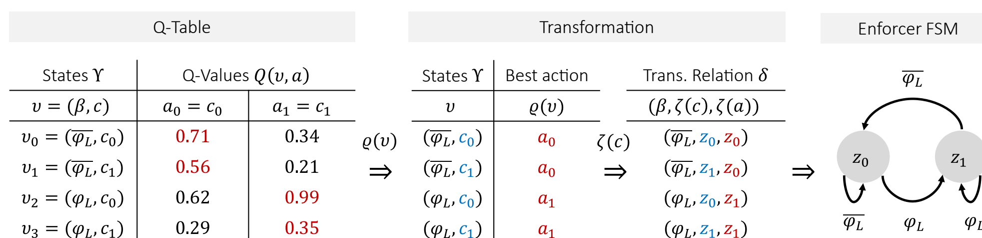

Figure 3 Example of transforming a Q-table that is based on the configuration set $C = \{c_0, c_1\}$ and the verification goal $VG_L := \mathcal{S}_{=?}[\varphi_L]$ for the latency requirement φ_L into an enforcer FSM.

Example. Let us give a simple example with two configurations $C = \{c_0, c_1\}$ and one verification goal $VG_L := \mathcal{S}_{=?}[\varphi_L]$ based on the latency requirement φ_L. As in the following Section 4 Evaluation, Q-learning is used as the reinforcement learning implementation. The Q-table of the enforcement agent contains one Q-value $Q(\upsilon, a)$ per tuple of enforcer agent state υ and action $a \in A = C$. The transformation from a Q-table into an enforcer FSM is illustrated in Fig. 3. In the first step, the transformation procedure associates each state υ with its best action a by applying ϱ. For Q-learning, the best action per state is directly given by the highest Q-value in the corresponding row, which therefore defines ϱ as:

$$\varrho(\upsilon) = \arg\max_{a \in A} \; Q(\upsilon, a). \tag{14}$$

With that, the transition relation δ can be determined, representing the two-step enforcer FSM depicted on the right side of the figure. Remember that each tuple $(\beta, z, z') \in \delta$ describes a transition of the enforcer FSM from state z into state z' triggered by the requirement response β. Finally, the resulting enforcer FSM can be formally verified for the given verification goals according to [10] using probabilistic model checking, e.g., PRISM [23].

4 Evaluation

In this section, we present an elaborate case study for the presented approach. The application considered is an object detection application whose actor graph is shown in Fig. 4 (in the appendix). The object detection application processes a stream of periodic input images in a pipelined fashion so that a given object in each image frame is detected based on scale-invariant feature transform (SIFT) matching [24]. As properties of execution to be enforced, we consider the latency o_L and the power consumption o_P. An empirical analysis of the execution times revealed that most of the execution time is spent in the SIFT description actor. Each SIFT description worker actor iterates over the list of features received from the control mechanism and generates corresponding feature descriptions, which is a compute-intensive task. For that reason, we apply RRE to just this actor using the method described in this paper. For the experiments, we used a sequence of $I_{train} = 1000$ images from the KITTI database [13] in each training episode. To counteract any violation of the corresponding requirements, a set of $m = 20$ power modes driven by dynamic voltage and frequency scaling (DVFS) can be applied, and a maximum of $n = 4$ cores can be allocated per run.

Each of the reinforcement learning-based enforcer instances was trained for a given set of verification goals until convergence (3,000 episodes, where each episode consists of iterating over the training input set I_{train}) using the reward function described in Eq. (6), the ϵ-greedy policy described in Eq. (4), parametrized with a learning rate of $\alpha = 0.1$ and a discount factor $\kappa = 0.99$. Due to the limited state space with $|C| = 80$, Q-learning is still viable. Note that

deep learning-based agent implementations such as Soft-Actor-Critic provide similar results but introduce additional overhead during training. The result of the transformation process, explained in Section 3.2, is an FSM that can be verified using the verification method in [10].

For formal verification, an environment FSM is generated from I_{train}, using the environment FSM generation method in [10], based on a latency requirement $\varphi_L = \varphi_L^{LB} \wedge \varphi_L^{UB} = (LB_{o_L} \leq o_L) \wedge (o_L \leq UB_{o_L})$ for a latency lower bound $LB_{o_L} = 0$ ms and an upper bound (deadline) $UB_{o_L} = 40$ ms, similarly for a power requirement $\varphi_P = \varphi_P^{LB} \wedge \varphi_P^{UB} = (LB_{o_P} \leq o_P) \wedge (o_P \leq UB_{o_P})$ for a power lower bound LB_{o_P} =0 W and an upper bound $UB_{o_P} =$ 1.2 W. Intuitively, $\varphi_L^{LB} = (0 \text{ ms} \leq o_L)$ and $\varphi_P^{LB} = (\ 0\,\text{W} \leq o_P)$ are always satisfied and can therefore be ignored during enforcement in this case study. The generated enforcer FSMs are verified using the PRISM model checker [22]. In the following, we present the verification results for loose and strict enforcement for the proposed and the following set of other previously proposed RRE techniques: Race-to-idle (RTI) [21] that executes the application in each iteration k constantly with $n = 4$ cores and the highest power mode m, 1-step enforcement FSM F_1 proposed in [9], and 8-step enforcement FSM F_2 in [10].

4.1 Loose enforcement

As a first example, we specify and verify the following two verification goals: $P_{=?}[G^{\leq 3}\neg\varphi_L]$ for the latency requirement φ_L and $P_{=?}[G^{\leq 3}\neg\varphi_P]$ for the power requirement φ_P, see Section 2.1.2 for explanation.

Table 1 Verification results for loose enforcement for RTI, F_1, F_2, and F_{rl_0} for the verification goals $P_{=?}[G^{\leq 3}\neg\varphi_L]$ and $P_{=?}[G^{\leq 3}\neg\varphi_P]$, based on a latency upper bound (deadline) $UB_{o_L} = 40$ ms, and a power upper bound $UB_{o_P} = 1.2$ W.

$P_{=?}[G^{\leq 3}\neg\varphi_L]$				$P_{=?}[G^{\leq 3}\neg\varphi_P]$			
RTI	F_1	F_2	F_{rl_0}	RTI	F_1	F_2	F_{rl_0}
0	0.427	0.041	0	1	0.256	0.389	0

As shown in Table 1, $P_{=?}[G^{\leq 3}\neg\varphi_L] = 0, P_{=?}[G^{\leq 3}\neg\varphi_P] = 0$ for the RL-generated FSM F_{rl_0}. This means that our approach can determine an enforcer FSM that does not violate the latency nor the power requirement for $\lambda = 3$ consecutive executions. $P_{=?}[G^{\leq 3}\neg\varphi_L] = 0$ also for RTI, as it always satisfies the latency requirement φ_L, and $P_{=?}[G^{\leq 3}\neg\varphi_L] = 1$ because it always violates the power requirement φ_P as it always runs in the highest power mode $m_{\max} = 20$ and number of cores $n_{\max} = 4$. Also note that $P_{=?}[G^{\leq 3}\neg\varphi_L]$ is higher for F_1 than for F_2 as F_2 increases its configuration state z by 8 steps when having a latency violation, whereas F_1 only increases it by 1. For the same reason, $P_{=?}[G^{\leq 3}\neg\varphi_P]$ for F_1 is lower than for F_2.

Finally, we also performed a verification for loose enforcement using the two alternative verification goals $S_{=?}[\neg\varphi_L]$ for the latency requirement φ_L and $S_{=?}[\neg\varphi_P]$ for the power requirement φ_P, see Section 2.1.2. Such steady-state probabilities give insight into the long-term behavior of running applications.

As shown in Table 2, the steady-state probabilities $S_{=?}[\neg\varphi_L]$ and $S_{=?}[\neg\varphi_P]$ for our RL-based FSM F_{rl_1} are lower than for F_1. Thus, our approach can generate an enforcer FSM that has lower steady-state probability than F_1 to violate the given requirements φ_L and φ_P. Although the steady-state probability of having a latency violation $S_{=?}[\neg\varphi_L]$ is lower for F_2 than F_{rl_1}, the steady-state probability of having a power violation $S_{=?}[\neg\varphi_P]$ for F_2 is higher than for F_{rl_1}. For RTI, as it always satisfies the latency requirement φ_L, it has also a stationary probability $S_{=?}[\neg\varphi_L] = 0$. But regarding the power requirement φ_P, $S_{=?}[\neg\varphi_L] = 1$

Table 2 Verification results for loose enforcement for RTI, F_1, F_2, and F_{rl_1} for the verification goals $S_{=?}[\neg\varphi_L]$ and $S_{=?}[\neg\varphi_P]$, based on a latency upper bound (deadline) $UB_{o_L} = 40$ ms, and a power upper bound $UB_{o_P} = 1.2\,\text{W}$.

$S_{=?}[\neg\varphi_L]$				$S_{=?}[\neg\varphi_P]$			
RTI	F_1	F_2	F_{rl_1}	RTI	F_1	F_2	F_{rl_1}
0	0.5	0.121	0.173	1	0.445	0.591	0.435

for RTI as it always runs in the power requirement violating mode $(n_{\max}, m_{\max})$. $S_{=?}[\neg\varphi_L]$ for F_2 is lower than for F_1 as F_2 increases its configuration state z by 8 when having a latency violation, whereas F_1 only increases it by 1, so it has a higher chance to meet the latency requirement φ_L. For the same reason, $S_{=?}[\neg\varphi_P]$ for F_1 is lower than for F_2.

4.2 Strict enforcement

Finally, the following example is chosen to illustrate the approach also for strict enforcement. Table 3 shows the verification results for strict enforcement of the latency requirement φ_L using the verification goal $AG(\varphi_L)$, see Section 2.1.2. $AG(\varphi_L) = \text{true}$ for RTI, which means that φ_L always holds, as RTI always runs in the highest power mode $m_{\max} = 20$ and number of cores $n_{\max} = 4$. For F_1 and F_2, $AG(\varphi_L) = \text{false}$ because both FSMs decrease their configuration state z by one once satisfying φ_L. Finally, our RL-based approach can generate an enforcer FSM F_{rl_2} that also always satisfy the latency requirement φ_L $(AG(\varphi_L) = \text{true})$.

Table 3 Verification results for strict enforcement for RTI, F_1, F_2, and F_{rl_2} for the verification goal $AG(\varphi_L)$, based on a latency upper bound (deadline) $UB_{o_L} = 40$ ms, and a power upper bound $UB_{o_P} = 1.2\,\text{W}$.

RTI	F_1	F_2	F_{rl_2}
true	false	false	true

5 Related Work

Several approaches do exist to control non-functional properties of program executions, such as latency, or power and energy consumption. Examples of such approaches are techniques based on online machine learning like [25, 26], heuristics like [42], and predictive models [8, 32]. However, most of them cannot provide any formal guarantees about the controller's capability of satisfying the given requirements. Such guarantees include that the control technique will never lead to a violation of the given requirements or that the system will stay no more than a certain number of executions in a violating state, or long-term percentages of non-violating executions. Although techniques based on control theory, such as [19, 27, 28], can formally analyze controller properties such as stability, they are not able to provide any formal guarantees regarding the satisfaction or violation of given non-functional requirements in uncertain environments.

In general, FSMs are not only used to formally specify the functional behavior of a system [30, 4, 12], but also when formal verification of non-functional properties is required, especially in safety-critical systems. In [40], the concept of *Run-time Requirement Enforcement (RRE)* is introduced to describe techniques to either centrally or decentrally control the satisfaction of non-functional execution properties of programs executed on MPSoCs given

by set of requirements. of programs for MPSoCs. Based on this concept, [10] proposes feedback-based RRE techniques. Presented is an approach for the formal specification and verification of non-functional properties for systems executing programs periodically, where an FSM-based enforcer is used to control the number of cores and DVFS level of a system once per execution at run-time. Using this approach, one can evaluate whether a combination of a system (MPSoC), an enforcer, and an environment either always satisfies the defined requirements or with which satisfaction probability. In [9], simple FSM control schemes are introduced that simply increase, resp. decrease the power mode or number of cores in case of a violation $\neg\varphi_L$, resp. satisfaction φ_L of the latency requirement. In [10, 11], FSMs for multi-requirement control have been introduced. However, the RRE controllers presented in these works are all manually designed. In summary, techniques for automatic generation of verifiable enforcers are still missing.

Reinforcement Learning techniques offer the capability for a controller to learn how to act for meeting run-time requirements via trials-and-errors on simulated or real data. There already exist several approaches to learn control techniques which leverage RL. Most of them use Q-Learning [44, 15], sometimes on a dedicated hardware module [7]. The majority of works that consider the verification of a trained RL policy are based on (DQL). Verification on DQL is intrinsically complex, partly due to the continuous input leading to an infinite number of states. [3] verifies DQL by extracting decision tree policies from a trained neural network. Though, they only verify robustness, stability and functional correctness of a controller. The authors of [20] and [18] both propose a *verification-in-the-loop* method, i.e., they perform the verification during training. However, their works only consider verification goals formulated using *ACTL* (a subset of CTL) and LTL specifications. Our approach, on the other hand, can verify goals formulated in not only CTL and LTL, but PCTL, too.

6 Conclusion

In this paper, we have presented a novel technique using reinforcement learning for automatically generating feedback-based run-time requirement enforcers that can be formally verified concerning a given set of verification goals by a model checker. For that, we elucidated a formalism for transforming reinforcement learning agents during training into enforcement state machines and applying model checking techniques in regular intervals to verify these to satisfy a set of verification goals. We were able to demonstrate in a case study using an object recognition application that our proposed approach significantly outperforms related work in being able to generate verified enforcers.

References

1 Nidhi Anantharajaiah, Tamim Asfour, Michael Bader, Lars Bauer, Jürgen Becker, Simon Bischof, Marcel Brand, Hans-Joachim Bungartz, Christian Eichler, Khalil Esper, Joachim Falk, Nael Fasfous, Felix Freiling, Andreas Fried, Michael Gerndt, Michael Glaß, Jeferson Gonzalez, Frank Hannig, Christian Heidorn, Jörg Henkel, Andreas Herkersdorf, Benedict Herzog, Jophin John, Timo Hönig, Felix Hundhausen, Heba Khdr, Tobias Langer, Oliver Lenke, Fabian Lesniak, Alexander Lindermayr, Alexandra Listl, Sebastian Maier, Nicole Megow, Marcel Mettler, Daniel Müller-Gritschneder, Hassan Nassar, Fabian Paus, Alexander Pöppl, Behnaz Pourmohseni, Jonas Rabenstein, Phillip Raffeck, Martin Rapp, Santiago Narváez Rivas, Mark Sagi, Franziska Schirrmacher, Ulf Schlichtmann, Florian Schmaus, Wolfgang Schröder-Preikschat, Tobias Schwarzer, Mohammed Bakr Sikal, Bertrand Simon, Gregor Snelting, Jan Spieck, Akshay Srivatsa, Walter Stechele, Jürgen Teich, Isaías A. Comprés Ureña, Ingrid Verbauwhede, Dominik Walter, Thomas Wild, Stefan Wildermann, Mario Wille, Michael Witterauf, and Li Zhang. *Invasive Computing*. FAU University Press, 2022.

2 Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. On the Logical Characterisation of Performability Properties. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 780–792. Springer, 2000.

3 Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable Reinforcement Learning via Policy Extraction. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 2499–2509, Red Hook, NY, USA, 2018. Curran Associates Inc.

4 Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield Synthesis. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 533–548. Springer, 2015.

5 Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

6 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, pages 39–48. IEEE, 2013.

7 Yvan Debizet, Guénolé Lallement, Fady Abouzeid, Philippe Roche, and Jean-Luc Autran. Q-Learning-based Adaptive Power Management for IoT System-on-Chips with Embedded Power States. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

8 Christophe Dubach, Timothy M Jones, Edwin V Bonilla, Michael FP O'Boyle, et al. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *MICRO*, volume 43, pages 485–496, 2010.

9 Khalil Esper, Stefan Wildermann, and Jürgen Teich. A comparative evaluation of latency-aware energy optimization approaches in many-core systems. In *Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

10 Khalil Esper, Stefan Wildermann, and Jürgen Teich. Enforcement FSMs: Specification and Verification of Non-Functional Properties of Program Executions on MPSoCs. In *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 21–31, 2021.

11 Khalil Esper, Stefan Wildermann, and Jürgen Teich. Multi-Requirement Enforcement of Non-Functional Properties on MPSoCs Using Enforcement FSMs-A Case Study. In *Third Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

12 Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime Enforcement Monitors: Composition, Synthesis, and Enforcement Abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

13 Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision Meets Robotics: The KITTI Dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

14 Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-Based Acceleration. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2829–2838. JMLR.org, 2016.

15 Ujjwal Gupta, Sumit K. Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y. Ogras. A Deep Q-Learning Approach for Dynamic Management of Heterogeneous Processors. *IEEE Computer Architecture Letters*, 18(1):14–17, 2019.

16 Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

17 Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

18 Mohammadhosein Hasanbeig, Daniel Kroening, and Alessandro Abate. Towards verifiable and safe model-free reinforcement learning. In Nicola Gigante, Federico Mari, and Andrea Orlandini, editors, *Proceedings of the 1st Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, co-located with the 18th International Conference of the Italian Association for Artificial Intelligence, OVERLAY@AI*IA 2019, Rende, Italy, November 19-20, 2019*, volume 2509 of *CEUR Workshop Proceedings*, page 1. CEUR-WS.org, 2019.

19 Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. POET: A Portable Approach to Minimizing Energy under Soft Real-Time Constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86. IEEE, 2015.

20 Peng Jin, Jiaxu Tian, Dapeng Zhi, Xuejun Wen, and Min Zhang. Trainify: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, volume 13371, pages 193–218. Springer International Publishing, Cham, 2022. Series Title: Lecture Notes in Computer Science. URL: `https://link.springer.com/10.1007/978-3-031-13185-1_10`.

21 David H. K. Kim, Connor Imes, and Henry Hoffmann. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications, CPSNA 2015, Kowloon, Hong Kong, China, August 19-21, 2015*, pages 78–85. IEEE Computer Society, 2015.

22 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Quantitative Analysis With the Probabilistic Model Checker PRISM. *Electron. Notes Theor. Comput. Sci.*, 153(2):5–31, 2006.

23 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

24 David G Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. IEEE, 1999.

25 Sumit K Mandal, Ganapati Bhat, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y Ogras. An Energy-Aware Online Learning Framework for Resource Management in Heterogeneous Platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 25(3):1–26, 2020.

26 Sumit K. Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y. Ogras. Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning. *IEEE Trans. Very Large Scale Integr. Syst.*, 27(12):2842–2854, 2019.

27 Anway Mukherjee and Thidapat Chantem. Energy Management of Applications With Varying Resource Usage on Smartphones. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11):2416–2427, 2018.

28 Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 174:1–174:9. ACM, 2013.

29 Behnaz Pourmohseni, Michael Glaß, Jörg Henkel, Heba Khdr, Martin Rapp, Valentina Richthammer, Tobias Schwarzer, Fedor Smirnov, Jan Spieck, Jürgen Teich, et al. Hybrid Application Mapping for Composable Many-Core Systems: Overview and Future Perspective. *Journal of Low Power Electronics and Applications*, 10(4):38, 2020.

30 Fred B Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

31 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint*, 2017. `arXiv:1707.06347`.

32 David C Snowdon, Etienne Le Sueur, Stefan M Petters, and Gernot Heiser. Koala: A Platform for OS-Level Power Management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, 2009.

33 Jan Spieck, Stefan Wildermann, and Jürgen Teich. Scenario-Based Soft Real-Time Hybrid Application Mapping for MPSoCs. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

34 Jan Spieck, Stefan Wildermann, and Jürgen Teich. Domain-Adaptive Soft Real-Time Hybrid Application Mapping for MPSoCs. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE, 2021.

35 Jan Spieck, Stefan Wildermann, and Jürgen Teich. A Learning-Based Methodology for Scenario-Aware Mapping of Soft Real-Time Applications onto Heterogeneous MPSoCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2022.

36 Jan Spieck, Stefan Wildermann, and Jürgen Teich. On Transferring Application Mapping Knowledge Between Differing MPSoC Architectures. In *CODES+ISSS 2022*, 2022.

37 Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. Adaptive computation and machine learning. MIT Press, 1998. URL: `https://www.worldcat.org/oclc/37293240`.

38 Jürgen Teich, Michael Glaß, Sascha Roloff, Wolfgang Schröder-Preikschat, Gregor Snelting, Andreas Weichslgartner, and Stefan Wildermann. Language and Compilation of Parallel Programs for-Predictable MPSoC Execution Using Invasive Computing. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 313–320. IEEE, 2016.

39 Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive Computing: An Overview. *Multiprocessor System-on-Chip*, pages 241–268, 2011.

40 Jürgen Teich, Pouya Mahmoody, Behnaz Pourmohseni, Sascha Roloff, Wolfgang Schröder-Preikschat, and Stefan Wildermann. Run-Time Enforcement of Non-functional Program Properties on MPSoCs. In *A Journey of Embedded and Cyber-Physical Systems*, pages 125–149. Springer, 2021.

41 Jürgen Teich, Behnaz Pourmohseni, Oliver Keszöcze, Jan Spieck, and Stefan Wildermann. Run-Time Enforcement of Non-Functional Application Requirements in Heterogeneous Many-Core Systems. In *25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020, Beijing, China, January 13-16, 2020*, pages 629–636. IEEE, 2020.

42 Xiaohang Wang, Amit Kumar Singh, Bing Li, Yang Yang, Hong Li, and Terrence Mak. Bubble Budgeting: Throughput Optimization for Dynamic Workloads by Exploiting Dark Cores in Many Core Systems. *IEEE Transactions on Computers*, 67(2):178–192, 2017.

43 Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

44 Amir Yeganeh-Khaksar, Mohsen Ansari, Sepideh Safari, Sina Yari-Karin, and Alireza Ejlali. Ring-DVFS: Reliability-Aware Reinforcement Learning-Based DVFS for Real-Time Embedded Systems. *IEEE Embedded Systems Letters*, 13(3):146–149, 2021.

A Appendix

Algorithm 1 Q-Learning.

Initialize Q arbitrarily
for all $ep \in \{1, 2, ..., n_{\text{episodes}}\}$ **do**
  Initialize υ arbitrarily
  while υ is not terminal **do**
    Pick an action a following an ϵ-greedy policy
    Take action a
    $(\upsilon', \xi) \leftarrow ObserveEnvironment(\upsilon, a)$
    $Q(\upsilon, a) \leftarrow UpdateQTable(\upsilon, \upsilon', a, \xi)$
    $\upsilon \leftarrow \upsilon'$
  end while
end for

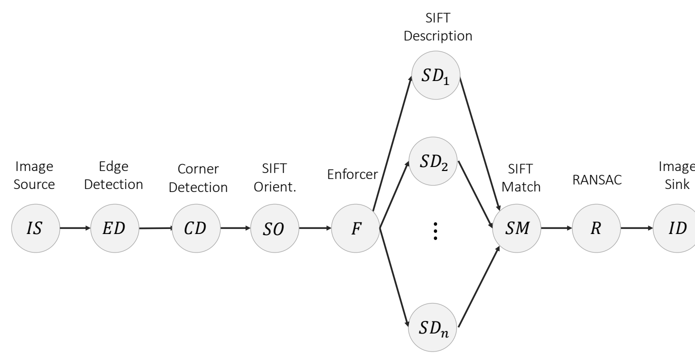

Figure 4 Actor graph of the evaluated object detection application.