A SBP PYTHON OBJECTS IMPLEMENTATION

The Python implementation of the three scenarios used in this paper is shown below: the code for *avoid back-and-forth rotation* appears in Fig. 8, the code for *avoid turns larger than 180°* appears in Fig. 9, and the code for *avoid turning when clear* appears in Fig. 10.

```
def SBP_avoidBackAndForthRotation():
    blockedEvList = []
    waitforEvList = [BEvent("SBP_MoveForward"),
                     BEvent("SBP_TurnLeft"),
                     BEvent("SBP_TurnRight")]
    while True:
        lastEv = yield {waitFor: waitforEvList, block: blockedEvList}
        if lastEv != BEvent("SBP_TurnLeft")
            and lastEv != BEvent("SBP_TurnRight"):
            blockedEvList = []
        else:
            blocked_ev = BEvent("SBP_TurnRight")
                if lastEv == BEvent("SBP_TurnLeft")
                else BEvent("SBP_TurnLeft")
            # Blocking!
            blockedEvList.append(blocked_ev)
```

Figure 8: The Python implementation of scenario *avoid back-and-forth rotation*. The code waits for any of the possible events: *SBP_MoveForward, SBP_TurnLeft* and *SBP_TurnRight*. Upon receiving *SBP_TurnLeft*, it blocks *SBP_TurnRight*, and upon receiving *SBP_TurnRight*, it blocks *SBP_TurnLeft*. Upon receiving *SBP_MoveForward*, it clears any blocking.

```
def SBP_avoid_k_consecuative_turns():
    k = 7
    counter = 0
    prevEv = None
    blockedEvList = []
    waitforEvList = [BEvent("SBP_MoveForward"), BEvent("SBP_TurnLeft"), \\
    BEvent("SBP_TurnRight")]
    while True:
        lastEv = yield {waitFor: waitforEvList, block: blockedEvList}
        if prevEv is None or lastEv == BEvent("SBP_MoveForward") or prevEv != lastEv:
            prevEv = lastEv
            counter = 0
            blockedEvList = []
        else:
            if counter == k - 1:
                # Blocking!
                blockedEvList.append(lastEv)
            else:
                counter += 1
```

Figure 9: The Python implementation of a scenario that blocks turning in the same direction more than k consecutive times. Each turn action rotates the robot by 30°, and so we set k to be 7.

```
def SBP_avoid_turning_when_clear():
    blockedEvList = []
    waitforEvList = [BEvent("SBP_MoveForward"), BEvent("SBP_TurnLeft"),\\
    BEvent("SBP_TurnRight")]
    while True:
        lastEv = yield {waitFor: waitforEvList, block: blockedEvList}
        state = lastEv.data['state']
        if state[3] > MINIMAL_FWD_CLEARANCE and state[2] > MINIMAL_CLEARANCE and \\
                state[4] > MINIMAL_CLEARANCE and abs(FWD_DIR - state[-2]) < FWD_DIR_TOLERANCE:
            blockedEvList.extend([BEvent("SBP_TurnLeft"), BEvent("SBP_TurnRight")])
        else:
            blockedEvList = []
```

Figure 10: The Python implementation of a scenario that blocks turning if the target is straight ahead and the path toward it is clear. The event carries data with it, which includes readings from the seven lidar sensors — with state[3] being the front-heading sensor. State[-2] is the direction to the target.

B FORMAL VERIFICATION

B.1 FORMAL VERIFICATION OF DNNS AND DRLS

A DNN verification algorithm receives the following inputs (Katz et al., 2017): a trained DNN N, a precondition P on the DNN's inputs, and a postcondition Q on N's output. The precondition is used to limit the input assignments to inputs of interest, or to express some assumption the user has regarding the environment (e.g., that an image-recognition DNN will only be presented with certain pixel values). The postcondition typically encodes the *negation* of the behavior we would like N to exhibit on inputs that satisfy P. Then, the verification algorithm searches for an input x' that satisfies the given conditions (i.e., $P(x') \wedge Q(N(x'))$), and returns exactly one of the following outputs: (i) SAT, indicating the query is satisfiable. Due to the postcondition Q encoding the negation of the required property, this result indicates that the wanted property is violated in some cases. Modern verification engines also supply a concrete input x' that satisfies the query, and hence, a valid input that triggers a bug, such as an incorrect classification; or (ii) UNSAT, indicating that there does not exist such an x', and thus — that the desired property always holds.

For example, suppose we wish to guarantee that for all non-negative inputs $x = \langle v_1^1, v_1^2 \rangle$, the DNN in Fig. 11 always outputs a value strictly smaller than 40; i.e., that that $N(x) = v_4^1 < 40$. This property can be encoded as a verification query consisting of a precondition that restrict the inputs to the desired range, i.e., $P = (v_1^1 \geq 0) \wedge (v_1^2 \geq 0)$, and by setting $Q = (v_4^1 \geq 40)$, which is the *negation* of the desired property. In this case, a sound verifier will return SAT, alongside a feasible counterexample such as $x = \langle 2, 3 \rangle$, which produces the output $v_4^1 = 48 \geq 40$ when fed to the DNN. Hence, the property does not always hold.

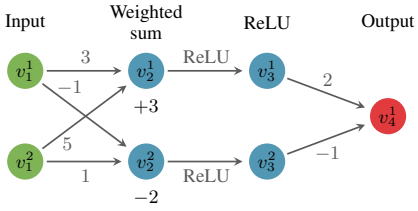


Figure 11: A toy DNN.

Originally, DNN verification engines were designed the verify the correct behaviour of feed-forward DNNs (Katz et al., 2017; Gehr et al., 2018; Wang et al., 2018; Lyu et al., 2020; Huang et al., 2017). However, in recent years, the verification community has also designed verification methods tailored for DRL systems (Corsi et al., 2021; Bacci et al., 2021; Eliyahu et al., 2021; Amir et al., 2021; 2022). These methods include techniques for encoding multiple invocations of the agent in question, when interacting with a reactive environment over multiple time-steps.

B.2 FORMAL VERIFICATION EXPERIMENTS

As an additional means of proving the effectiveness of our method, we ran formal verification queries relating to the aforementioned undesirable behaviors. In order to conduct a fair comparison, we selected only models that passed our success cutoff value (85%); and for each of these models we ran three verification queries — each checking whether the model violates a given property (SAT),

	avoid back-and-forth rotation			*avoid turns larger than 180°*			*avoid turning when clear*		
ALGO	SAT	UNSAT	TIMEOUT	SAT	UNSAT	TIMEOUT	SAT	UNSAT	TIMEOUT
Baseline	60	0	0	51	0	9	60	0	0
SBP	22	38	0	0	41	19	9	34	17

Table 1: Results of the formal verification queries over a total of 120 trained DNNs, for each of the three properties in question. The first row shows the results of the 60 baseline policies, and the second row shows results of the 60 policies trained by our method, with all rules active.

or abides by it for all inputs (UNSAT). We note that a verifier might also fail to terminate, due to TIMEOUT or MEMOUT errors. Each query ran with a TIMEOUT value of 36 hours, and a MEMOUT value of 6 GB. Table 1 summarizes the results of our experiments.

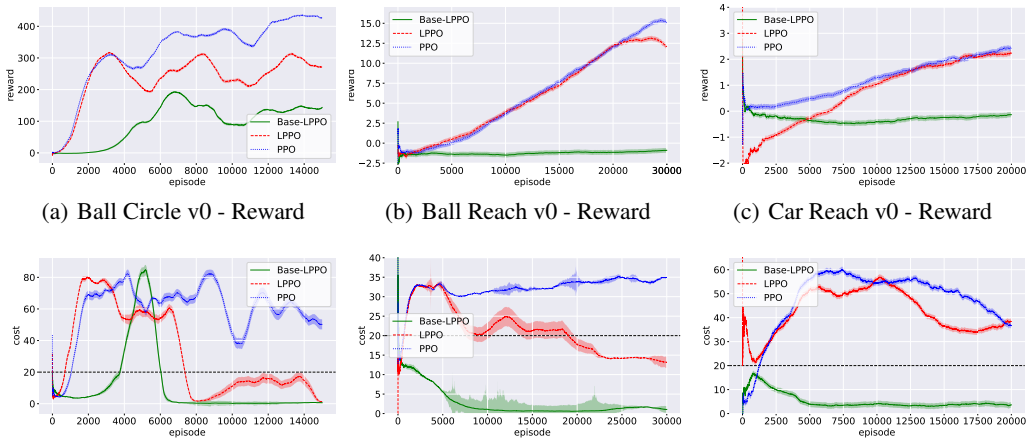
These results show a *significant* change of behavior between DNNs trained with the baseline algorithm, and those trained by our method. Indeed, we see that the latter policies much more often completely abide by the specific rules, and are consequently far more reliable.

C ANALYSIS ON STANDARD BENCHMARK

To further validate our method, we provide a more intensive study of our optimized implementation of the Lagrangian PPO (detailed in Section 4), without the additional rules (e.g., the SBP Scenarios). We perform our analysis on the standard benchmark *Bullet Safety Gym* (Gronauer, 2021). Bullet Safety Gym is an open source suite of different environments based on *PyBullet* and the most updated versions of Python, which implements the standard environments from *SafetyGym* (Ray et al., 2019), adding more robots and tasks. The crucial feature of the environments from Bullet Safety Gym is that they implement a Constrained Markov Decision Process (CMDP). In these environments, the objective is to maximize the reward function and maintain a cost function below a given threshold. We refer to the main paper from Ray et al. (2019) for more details. We selected a subset of environments for our analysis: *Ball Circle v0*, *Ball Reach v0* and *Car Reach v0*. A detailed description of the environments can be found in the open source repository of Bullet Safety Gym (Gronauer, 2021).

Fig. 12 shows our results on the three environments, comparing cost and reward functions obtained with (i) the standard PPO algorithm (*PPO*); (ii) a standard Lagrangian PPO (*Base-LPPO*); and (iii) our optimized Lagrangian PPO (*LPPO*) described in Section 4 of the main paper.

Overall, these results show that our algorithm can get similar performance reward-wise as *PPO*, and can get the cost below the required threshold in two out of three test cases. However, in Fig. 12(a), on the *Ball Circle v0* environment, *PPO* reaches significantly better reward-wise performance than ours. Moreover, in Fig. 12(f), in the *Car Reach v0* environment, our algorithm fails to get the cost below the required threshold. We leave it for future work, to study the impact of adding expert-knowledge rules with our optimizations, and if those will enable obtaining good performance reward-wise and cost-wise, also in those cases, for the *Ball Circle v0* environment reward, shown in Fig. 12(a), and the *Car Reach v0* environment cost, shown in Fig. 12(f).



(a) Ball Circle v0 - Reward (b) Ball Reach v0 - Reward (c) Car Reach v0 - Reward

(d) Ball Circle v0 - Cost (e) Ball Reach v0 - Cost (f) Car Reach v0 - Cost

Figure 12: A comparison between the original PPO (Schulman et al., 2017) (*PPO*), a standard implementation of Lagrangian PPO (Ray et al., 2019) (*Base-LPPO*), and our optimized version of the Lagrangian PPO (*LPPO*). The analysis is performed on the standard benchmark *Bullet Safety Gym* (Gronauer, 2021), and in particular on three environments of the suite (*Ball Circle v0*, *Ball Reach v0* and *Car Reach v0*).

In summarizing, Fig. 12 shows that, not surprisingly, the standard PPO reaches good performance reward-wise but can not optimize the cost. A naïve implementation of the Lagrangian PPO can minimize the cost function but struggles to obtain good performance reward-wise.

Our optimized approach (without SBP rules) is the only one that, at the same time, succeeds in reducing the cost under the given threshold in two out of three cases while reaching good performance reward-wise. We are confident that we can define SBP rules that will help to reduce those costs as well as further improve the reward.

D SHORT INTRODUCTION TO SBP

Scenario-based programming (SBP) (Damm & Harel, 2001; Harel & Marelly, 2003) is a paradigm designed to facilitate the development of reactive systems by allowing engineers to program behaviors with a focus on inter-object system-wide behaviors. It can be regarded as a realization of the Live Sequence Charts (LSCs) formalism (Damm & Harel, 2001), which is a visual language for specifying scenarios. LSCs were initially used for requirement specification and later evolved into a full-blown programming language. Today, there exist multiple implementations of SBP, such as BPJ (BP in Java) (Harel et al., 2010), BP-Py (BP in Python) (Yaacov, 2020), and others, making SBP accessible to many programmers and engineers.

In SBP, a system is composed of *scenarios*, each describing a single, desired or undesired behavioral aspect of the system; and these scenarios are then executed together as a cohesive system. An execution of a scenario-based (SB) program is formalized as a discrete sequence of events. At each time-step, the scenarios synchronize with each other to determine the next event to be triggered. Each scenario declares events that it *requests* and events that it *blocks*, corresponding to desirable and undesirable (forbidden) behaviors from its perspective; and also events that it passively *waits-for*.

After making these declarations, the scenarios are temporarily suspended by the run-time engine, and an *event-selection mechanism* triggers a single event that was requested by at least one scenario and blocked by none. Scenarios that requested or waited for the triggered event wake up, perform local actions, and then synchronize again; and the process is repeated ad infinitum. The resulting execution thus complies with the requirements and constraints of each of the individual scenarios (Harel & Marelly, 2003; Harel et al., 2012b). For a formal definition of SBP, see (Harel et al., 2012b).

Although SBP is implemented in many high-level languages, it is often convenient to think of scenarios as transition systems, where each state corresponds to a synchronization point, and each edge corresponds to an event that could be triggered. Fig. 13 uses state-transition representation to depict a simple SB program that controls the temperature and water-level in a water tank (borrowed from (Harel et al., 2012a)).

The scenarios *add hot water* and *add cold water* repeatedly wait for `WATER LOW` event, and then request three times the event `Add HOT` or `Add COLD`, respectively. Since these six events may be triggered in any order by the event selection mechanism, a new scenario *stability* is added to keep the water temperature stable, achieved by alternately blocking `Add HOT` and `Add COLD` events. The Python implementation code of this program, with the scenarios *add hot water*, *add cold water* and *stability*, appears in Fig. 14.

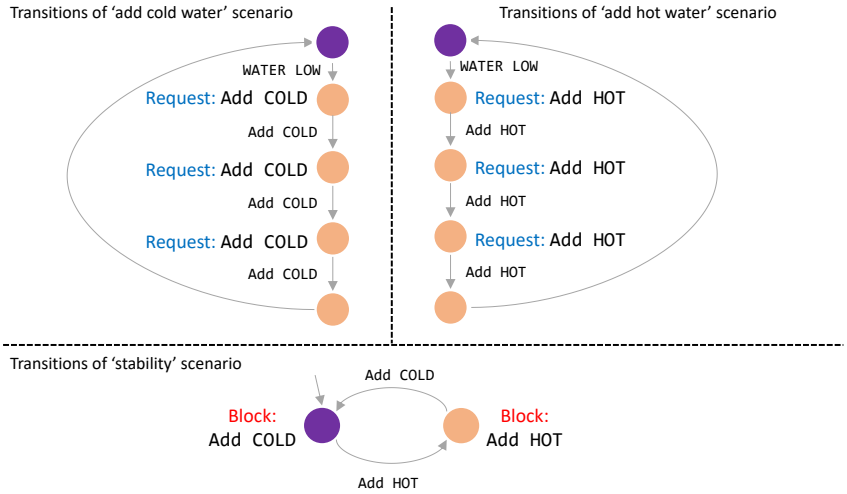


Figure 13: The state transition graphs represent the scenarios of a scenario-based program for controlling a water tank. The *add hot water* and *add cold water* scenarios wait in their initial state for a `WATER LOW` event. Once `WATER LOW` is triggered, they each move to their next state, requesting `Add Cold` and `Add Hot` events, respectively. The *stability* scenario waits in its initial state for a `Add Hot` event, while blocking `Add Cold` events. Once an `Add Hot` event is triggered, the scenario transitions to its second state, where it blocks `Add Hot` events while waiting for an `Add Cold` event. Once an `Add Cold` event is triggered, the scenario transitions back to its initial state, in which it waits for an `Add Hot` event while blocking `Add Cold` events.

```
def add_hot_water():
  while True:
    yield {waitFor: BEvent("WATER LOW")}
    yield {request: BEvent("Add HOT")}
    yield {request: BEvent("Add HOT")}
    yield {request: BEvent("Add HOT")}

def add_cold_water():
  while True:
    yield {waitFor: BEvent("WATER LOW")}
    yield {request: BEvent("Add COLD")}
    yield {request: BEvent("Add COLD")}
    yield {request: BEvent("Add COLD")}

def stability():
  while True:
    yield {waitFor: BEvent("Add HOT"), block: BEvent("Add COLD")}
    yield {waitFor: BEvent("Add COLD"), block: BEvent("Add HOT")}
```

Figure 14: The Python implementation of the three scenarios: *add hot water*, *add cold water*, and *stability*. The Python code will run until it reaches a synchronization point, indicated by a *yield* statement, where it will stop and declare events it waits for, requests, and blocks. Once an event that the scenario requested or waited for is triggered, the run-time engine will resume the scenario's execution; and the code will run until its next synchronization point, and repeat.