A Behavioral Programming Semantics

The semantics of behavioral programming were first presented in (Harel et al., 2010), defining the concept of b-threads and their composition using labeled transition systems (LTS). Recall that an LTS is defined as a tuple $\langle S, E, \rightarrow, init \rangle$, where S is a set of states, E is a set of events, $\rightarrow \subseteq S \times E \times S$ is a transition relation, and $init \in S$ is the initial state (Keller, 1976). The runs of such a transition system are sequences of the form $s^0 \xrightarrow{e^1} s^1 \xrightarrow{e^2} \ldots \xrightarrow{e^i} s^i \ldots$ where $s^0 = init$, and $\forall i = 1, 2, \ldots, s^i \in S$, $e^i \in E$, and $s^{i-1} \xrightarrow{e^i} s^i$.

Definition 1 (b-thread)**.** *A b-thread is a tuple* $\langle S, E, \rightarrow, init, B, R \rangle$ *where* $\langle S, E, \rightarrow, init \rangle$ *forms a labeled transition system,* $R\colon S \rightarrow 2^E$ *associates a state with the set of events requested by the b-thread in that state, and* $B\colon S \rightarrow 2^E$ *associates a state with the set of events blocked in that state.*

Example 1. The `add_hot` b-thread presented in Listing 1 consists of four states—one for each iteration i in the loop (s_1, s_2, s_3) and a terminal state (s_4) reached when the loop ends. At each iteration, the b-thread requests the `Hot` event and blocks no event. Formally, we get that $\forall i \in \{1,2,3\}\colon R(s_i) = \{Hot\}, B(s_i) = \emptyset$. Once the loop ends, the b-thread reaches the terminal state, s_4, where it does not submit statements to the event arbiter. Hence, trivially, we get that $B(s_4) = \emptyset, R(s_4) = \emptyset$.

Definition 2 (b-program)**.** *A b-program is a set of b-threads* $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i \rangle\}_{i=1}^n$

In line with the definitions of b-threads in (Harel et al., 2010), an abstraction is chosen in which a state of a b-thread is defined only at the synchronization points. This means that b-threads' activities between synchronization points are considered atomic and are independent of each other. We accord with (Harel et al., 2011), noting that this assumption is reasonable and does not unduly constrain the capabilities of b-programs.

Definition 3. *(run of a b-program). A run of a b-program,* $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i \rangle\}_{i=1}^n$*, is a run of the labeled transition system* $\langle S, E, \rightarrow, init \rangle$*, where* $S = S_1 \times \cdots \times S_n, E = \bigcup_{i=1}^n E_i, init = \langle init_1, \ldots, init_n \rangle$*, and* $\rightarrow$ *includes a transition* $\langle s_1, \ldots, s_n \rangle \xrightarrow{e} \langle s'_1, \ldots, s'_n \rangle$ *if and only if*

$$e \in \bigcup_{i=1}^{n} R_i(s_i) \bigwedge e \notin \bigcup_{i=1}^{n} B_i(s_i)$$

and

$$\bigwedge_{i=1}^{n} ((e \in E_i \Rightarrow s_i \xrightarrow{e} s'_i) \wedge (e \notin E_i \Rightarrow s_i = s'_i)).$$

In general, multiple runs of a b-program may exist, depending on the sequence in which events are selected from the set of requested and unblocked events (Harel et al., 2010).

B Translation of Behavioral Programs to PRISM

We have implemented an automatic translation process for BP programs to PRISM models, similar to the one described in Section 4. In Listing 10, we show an example of how the b-thread in Listing 5 of an uneven coin flip is translated into a module in PRISM. We model the b-thread as a state machine with four distinct states:

- State 0: The initial state.
- State 1: The state following the selection of 'heads' in the choice.
- State 2: The state after the selection of 'tails' in the random choice.
- State 3: The final state upon the completion of the b-thread's execution.

The variable `s_coin_flip` is defined at Line 6 to represent the current state of the b-thread. According to Line 8, if the b-thread is in State 0, it transitions to either State 1 or State 2 with respective probabilities of 0.4 and 0.6 without event synchronization. Lines 9 and 10 dictate that if the b-thread is in State 1, it either progresses to State 3 if the event `heads` is triggered or remains in State 1 if the event `tails` is triggered. Similarly, lines 11 and 12 describe the behavior of State 2. Lines 12 and 13 indicate that State 3 is a sink state, meaning that it remains there regardless of which event is triggered. In addition to this state machine, we model formulas in lines 1-5 that indicate whether each event is requested and/or blocked by the b-thread depending on its current state.

The main module, shown in Listing 11, implements the BP semantics by collecting all the requested and not blocked events and allowing a non-deterministic choice between them. This main module is somewhat degenerated since we only have one b-thread in the b-program. In bigger examples, each b-thread is modeled as a separate module, and the main module collects the information from the module to synchronize the collective non-deterministic choice of events.

```
formula coin_flip_req_heads = (s_coin_flip=2);
formula coin_flip_req_tails = (s_coin_flip=1);
formula coin_flip_block_heads = false;
formula coin_flip_block_tails = false;
module coin_flip
  s_coin_flip: [0..3] init 0;

  []      (s_coin_flip=0) -> 0.4: (s_coin_flip'=2) + 0.6: (
      s_coin_flip'=1);
  [heads] (s_coin_flip=1) -> 1: (s_coin_flip'=1);
  [tails] (s_coin_flip=1) -> 1: (s_coin_flip'=3);
  [heads] (s_coin_flip=2) -> 1: (s_coin_flip'=3);
  [tails] (s_coin_flip=2) -> 1: (s_coin_flip'=2);
  [heads] (s_coin_flip=3) -> 1: (s_coin_flip'=3);
  [tails] (s_coin_flip=3) -> 1: (s_coin_flip'=3);

endmodule
```

Listing 10: A translation of the `coin_flip` b-thread in Listing 5 to a PRISM module

```
formula heads_req = (coin_flip_req_heads=true);
formula tails_req=(coin_flip_req_tails=true);
formula heads_block=(coin_flip_block_heads=true);
formula tails_block = (coin_flip_block_tails=true);
formula heads_enabled = (heads_req=true) & (heads_block=
    false);
formula tails_enabled = (tails_req=true) & (tails_block=
    false);
module main
  event: [-1..2] init -1;
  [heads] (heads_enabled=true) -> 1: (event'=0);
  [tails] (tails_enabled=true) -> 1: (event'=1);
endmodule
```

Listing 11: The main module of the translation of the code given in Listing 5 to PRISM.

C DRL Evaluation Results

The results of the first evaluation for the *Cinderella-Stepmother Problem*, focused on the task of finding a deterministic event selection mechanism, are presented in Table 6. The experiment was conducted on an Intel Xeon E5-2620 CPU, with each measurement averaged over 10 repetitions. We tested the program with various parameter values, specifically $n \in \{5, 10, 15, 20\}$ and $b \in \{50, 100, 150, 200, 250, 300\}$. Similar to the results of the pancake example presented in Section 6, the runtime and memory of the synthesis approach increase with the problem size. In contrast, the DRL approach's runtime and memory remain relatively stable.

We continued to evaluate the task of finding a valid non-deterministic strategy, a common task in supervisory control aiming to reduce violations and achieve permissive behavior (Ramadge and Wonham, 1987). Using DRL, the network must distinguish between events leading to violations and those that do not. This is distinct from seeking an optimal deterministic policy that maximizes cumulative rewards, which is the focus of the PPO algorithm (an *on-policy* algorithm). Thus, we experimented with two different algorithms: the standard DQN (Mnih et al., 2013) and the Quantile Regression DQN (QR-DQN) (Dabney et al., 2018), implemented in (Hill et al., 2018), using an MLP network with two hidden layers of size 64. Both algorithms learn the maximal cumulative reward

		Time[1]		Memory[2]				Time[1]		Memory[2]	
n	b	**DRL**	**Syn.**	**DRL**	**Syn.**	n	b	**DRL**	**Syn.**	**DRL**	**Syn.**
5	50	22	3	2.16	0.04	15	50	21	9	2.16	0.05
	100	23	9	2.16	0.05		100	25	26	2.16	0.06
	150	28	19	2.16	0.06		150	43	57	2.16	0.08
	200	28	33	2.16	0.07		200	49	100	2.16	0.11
	250	22	51	2.16	0.08		250	52	155	2.16	0.15
	300	21	75	2.16	0.10		300	58	221	2.16	0.19
10	50	19	5	2.16	0.04	20	50	24	13	2.16	0.05
	100	32	17	2.16	0.05		100	24	37	2.16	0.07
	150	28	37	2.16	0.07		150	35	80	2.16	0.10
	200	37	65	2.16	0.09		200	60	139	2.16	0.14
	250	34	102	2.16	0.11		250	63	215	2.16	0.18
	300	31	143	2.16	0.14		300	51	307	2.16	0.23

[1] in seconds, [2] in GB

Table 6: Runtime and memory comparison between the DRL and program synthesis approaches in finding a single valid execution trace for the Cinderella-Stepmother Problem.

achievable for any given state-event pair (*off-policy* algorithm). The learned values are used to disable events below a predefined threshold during execution. Since we wish to eventually be able to add blueberries and receive a reward of 1, we set the threshold to 0. The ground truth strategy was computed by exploring the entire state space, classifying states leading to violations as red and the rest as green. The ability of the two algorithms to differentiate between good (visits only green states) and bad (visits some red states) sampled traces was assessed during learning.

Figure 7 depicts the precision and recall of the two algorithms in distinguishing between the sampled traces during the learning phase. In our context, precision represents the fraction of good traces among the traces the algorithm accepts as a part of its policy, and recall represents the fraction of accepted traces among the good traces in the entire sample. We observe that the DQN algorithm quickly learned a policy accepting about 87% of good traces but was unstable, allowing bad traces. In contrast, QR-DQN took more time to learn but was stable, consistently excluding bad traces from its policy.

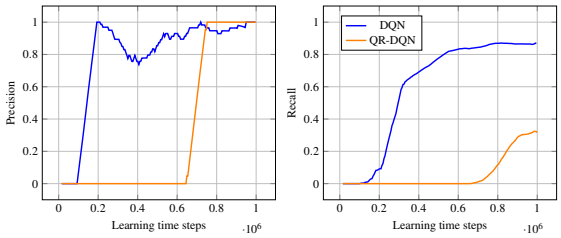

Figure 7: Average precision and recall of the DQN and QR-DQN learning algorithms in the pancake maker example with $n = 200$ and $b = 25$. A moving average window of 20 was applied due to the high variance of measurements.

The results of the second evaluation for the

Cinderella-Stepmother Problem are presented in Table 6. The experiment was conducted using an NVIDIA GeForce RTX 4090 GPU. Both algorithms reached relatively high and stable precision rapidly, with a slight advantage to the QR-DQN algorithm. We note that Table 6 presents only the precision since both algorithms stabilized at a recall of 1 in the initial measurements.

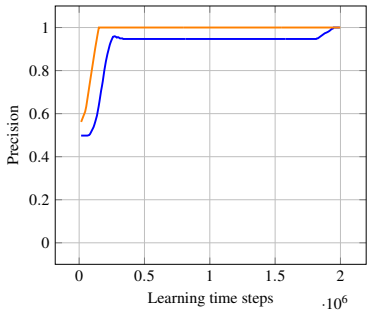

Figure 8: Average precision of the DQN and QR-DQN learning algorithms in the Cinderella-Stepmother Problem with $n = 5$ and $b = 50$. A moving average window of 20 was applied due to the high variance of measurements.

D BP ⇔ (DRL+Probabilities+SMT) Evaluation Program

The program used for the evaluation in Section 7 is outlined in Listing 12. It represents the matrix with the variable `p`. In each round, the row or column that is flipped in `p` is determined by the constraints defined by the `row_flip` and `col_flip` functions. The `opponent` first flips a random row and then flips a random column using `choice`. The player is modeled using the `controller` b-thread, which flips rows and columns based on the variable `action` controlled by the DRL mechanism. To specify the target optimization criteria, we used the `localReward` parameter in the `reward_bt` b-thread, which rewards exponentially based on the number of bits converted to positive in each round.

```
p=[[Bool(f"p{i}{j}") for j in range(M)] for i in range(N)]
action = Int("action")

@thread
def init():
  chess=And([p[i][j] if (i+j)%2 else Not(p[i][j]) for i in
      range(N) for j in range(M)])
  yield sync(request=chess)

@thread
def opponent():
  e = yield sync(waitFor=true)
  while True:
    i = choice({k: 1 / N for k in range(N)})
    yield sync(request=row_flip(i, e))
    e = yield sync(waitFor=true)
    j = choice({k: 1 / M for k in range(M)})
    yield sync(request=col_flip(j, e))
    e = yield sync(waitFor=true)

def row_actions(e):
  return And([Implies(action == i, row_flip(i, e)) for i in
      range(N)])

def col_actions(e):
  return And([Implies(action == i, col_flip(i, e)) for i in
      range(M)])

@thread
def controller():
  e = yield sync(waitFor=true)
  while True:
    e = yield sync(waitFor=true)
    yield sync(block=Not(row_actions(e)), waitFor=true)
    e = yield sync(waitFor=true)
    yield sync(block=Not(col_actions(e)), waitFor=true)

def count_reward(e_0, e_1):
    if e_0 is None or e_1 is None:
        return 0
    def turned_on(e_0, e_1, i, j):
        return is_true(e_1.eval(p[i][j])) and not is_true(
      e_0.eval(p[i][j]))
    return 2**sum([int(turned_on(e_0, e_1, i, j)) for i in
      range(N) for j in range(M)])

@thread
def reward_bt():
    e_0, e_1 = None, None
    while True:
        e_0, e_1 = e_1, yield sync(waitFor=true,
      localReward=count_reward(e_0, e_1))
```

Listing 12: A solver-based implementation for the bit-flip two-player game.

REFERENCES

Anderson, M. and Feil, T. (1998). Turning lights out with linear algebra. *Mathematics Magazine*.

Ashrov, A. and Katz, G. (2023). Enhancing Deep Learning with Scenario-Based Override Rules: a Case Study. In *MODELSWARD*.

Bar-Sinai, M. (2020). *Extending Behavioral Programming for Model-Driven Engineering*. PhD Thesis, Ben-Gurion University of the Negev, Israel.

Bar-Sinai, M. and Weiss, G. (2021). Verification of Liveness and Safety Properties of Behavioral Programs Using BPjs. In *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*.

Bar-Sinai, M., Weiss, G., and Shmuel, R. (2018). BPjs: an extensible, open infrastructure for behavioral programming research. In *MODELS*.

Bodlaender, M. H. L., Hurkens, C. A., Kusters, V. J., Staals, F., Woeginger, G. J., and Zantema, H. (2012). Cinderella versus the wicked stepmother. In *Theoretical Computer Science*.

Busard, S. and Pecheur, C. (2013). PyNuSMV: NuSMV as a Python Library. In *NASA Formal Methods*.

Dabney, W., Rowland, M., Bellemare, M., and Munos, R. (2018). Distributional Reinforcement Learning With Quantile Regression. *AAAI*.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *TACAS*.

Depuydt, L. and Gill, R. D. (2012). Higher Variations of the Monty Hall Problem (3.0 and 4.0) and Empirical Definition of the Phenomenon of Mathematics, in Boole's Footsteps, as Something the Brain Does. *Advances in Pure Mathematics*, 02.

Eitan, N. and Harel, D. (2011). Adaptive behavioral programming. In *ICTAI*.

Elyasaf, A. (2021). Context-Oriented Behavioral Programming. *Information and Software Technology*, 133.

Elyasaf, A., Sadon, A., Weiss, G., and Yaacov, T. (2019). Using Behavioural Programming with Solver, Context, and Deep Reinforcement Learning for Playing a Simplified RoboCup-Type Game. In *MODELS*.

Elyasaf, A., Yaacov, T., and Weiss, G. (2023). What Petri Nets Oblige us to Say: Comparing Approaches for Behavior Composition. *IEEE Transactions on Software Engineering*, 49.

Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., and Weiss, G. (2013). On composing and proving the correctness of reactive behavior. In *EMSOFT*.

Harel, D., Lampert, R., Marron, A., and Weiss, G. (2011). Model-checking behavioral programs. In *EMSOFT*.

Harel, D., Marron, A., and Weiss, G. (2010). Programming Coordinated Behavior in Java. In *ECOOP*.

Harel, D., Marron, A., and Weiss, G. (2012). Behavioral programming. *Communications of the ACM*, 55.

Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T., and Volk, M. (2022). The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*, 24.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable Baselines.

Huang, S. and Ontañón, S. (2022). A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. In *FLAIRS*.

Katz, G., Marron, A., Sadon, A., and Weiss, G. (2019). On-the-fly construction of composite events in scenario-based modeling using constraint solvers. In *MODELSWARD*.

Keller, R. M. (1976). Formal verification of parallel programs. *Communications of the ACM*, 19.

Knuth, D. (1976). The complexity of nonuniform random number generation. *Algorithm and Complexity, New Directions and Results*.

Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*.

McMillan, K. L. (1993). The SMV System. *Symbolic Model Checking*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint*.

Naveed, H., Arora, C., Khalajzadeh, H., Grundy, J., and Haggag, O. (2024). Model driven engineering for machine learning components: A systematic literature review. *Information and Software Technology*, 169.

Puterman, M. L. (1990). Markov decision processes. In *Stochastic Models*, volume 2. Elsevier.

Qin, X., Bliudze, S., Madelaine, E., Hou, Z., Deng, Y., and Zhang, M. (2020). Smt-based generation of symbolic automata. *Acta Informatica*, 57.

Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, 3.

Ramadge, P. J. and Wonham, W. M. (1987). Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 25.

Savitch, W. J. (1970). Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4.

Shevrin, I. and Yossef, M. (2020). Spectra example: Cinderella-stepmother problem.

Stoelinga, M. (2004). An Introduction to Probabilistic Automata. *Bulletin of the EATCS*, 78.

Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, A. J. S., and Younis, O. G. (2023). Gymnasium.

Yaacov, T. (2023). BPpy: Behavioral programming in Python. *SoftwareX*, 24.

Yaacov, T., Elyasaf, A., and Weiss, G. (2024). Keeping Behavioral Programs Alive: Specifying and Executing Liveness Requirements. In *International Requirements Engineering Conference*.

Yerushalmi, R., Amir, G., Elyasaf, A., Harel, D., Katz, G., and Marron, A. (2023). Enhancing Deep Reinforcement Learning with Scenario-Based Modeling. *SN Computer Science*, 4.