Ben-Gurion University of the Negev
The Faculty of Natural Sciences
The Department of Computer Science

Extending Behavioral Programming Towards Improved Software Engineering Practices

Thesis submitted in partial fulfillment of the requirements
for the Master of Sciences degree

Tom Yaacov

Under the supervision of Dr. Achiya Elyasaf and Prof. Gera Weiss

June 2021

Ben-Gurion University of the Negev
The Faculty of Natural Sciences
The Department of Computer Science

Extending Behavioral Programming Towards Improved Software Engineering Practices

Thesis submitted in partial fulfillment of the requirements
for the Master of Sciences degree

Tom Yaacov

Under the supervision of Dr. Achiya Elyasaf and Prof. Gera Weiss

Signature of student: _____ Date: _____
Signature of supervisor: _____ Date: _____
Signature of supervisor: _____ Date: _____
Signature of chairperson of the
committee for graduate studies: _____ Date: _____

June 2021

Abstract

This thesis presents several extensions of the Behavioral Programming (BP) paradigm to make it more accessible and usable for applications. Our contributions are as follows:

BP/Python: A stable, unified implementation of BP in Python. This implementation is used as an infrastructure for the research presented here and is already being used by others in our research group and outside of it.

BP/DRL: An experimental protocol for a more natural and abstract system modelling. In this protocol, a combination of BP and Deep Reinforcement Learning (DRL) allows for giving abstract instructions to a system and for teaching it to follow them.

BP/Liveness: A method for allowing direct specification of liveness requirements in BP. By integrating Reinforcement Learning to the event selection mechanism, we show that liveness requirements can be enforced in execution without adding unnecessary external constraints.

BP/DES: A methodology for Discrete Event Systems (DES) modelling using BP and Petri Net. The methodology offers to model system requirements with BP prior to implementing it with petri net, to avoid premature implementation decisions.

We show how combinations of these contributions improve software development processes by supporting a natural and incremental development of software systems. We compare BP with our improvements to the baseline BP and other development techniques. Many examples given in this thesis show that software developed with the proposed methods is better aligned with the specification of systems, as the programmer perceive them, and thus is easier to develop and maintain.

The quest for improving programming languages and methodologies is longstanding and is far from over. As we believe that the way ahead lies in integrating methods, our focus was on the integration of BP with other methods and tools, as apparent in the above list.

Keywords: Behavioral Programming, Python, Constraint Solver, Reinforcement Learning, Liveness, Discrete Event System, Petri Net.

Acknowledgements

First of all, I would like to express sincere gratitude to my advisors, Gera Weiss, and Achiya Elyasaf. I could not have made it through the long journey of writing this thesis without the advice, guidance, and inspiration I received from you.

I have collaborated and consulted with fellow researchers along the way. I thank Aviran Sadon for his contribution at different stages of the research presented here. Additionally, I thank Assaf Marron and Michael Bar-Sinai for their advisory. I would also like to acknowledge the Computer Science Department at Ben-Gurion University for providing a friendly and supportive environment.

I am also grateful to my family and friends, especially my parents, Arie and Ilana Yaacov, for encouraging me during the compilation of this thesis. Lastly, I want to thank my life partner, Rotem Shmueli, who motivated me and provided unconditional support along the journey.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Behavioral Programming paradigm aims at improving software development and modelling processes [1]. In this work, we enhance the paradigm and show how, with our enhancements, it can simplify and improve software development processes in various domains.

Software systems are becoming ever complex [2]. This complexity growth stems from the need for handling more complex requirements, as the saying goes "appetite comes with eating". This driving force challenges systems developers to explore and take their software development tools to their limits. With that said, as pointed out by Dalal [3], demand for a complex software system is increasing more rapidly than the ability to design, implement, test, and maintain them.

A wide range of paradigms have been devised to handle the complexity of software engineering development. The paradigms are designed to make the engineering process easier and/or extend the complexity of applications that can feasibly be built. While progress is constantly made, there is a wide agreement among researchers and practitioners that there is much more to do in this frontier [4].

The work presented here focuses on allowing programmers to focus on formulating aspects of the system behavior as they see them. Then, we show, that it is possible, using advanced techniques, to interleave these aspects together into a working application. We show that with better programming idioms, developers can harness the power of advanced algorithms and techniques by augmenting them with aspects of behaviors that they specify in a natural and intuitive way, thus, simplifying and enhancing programming. We consider all the different development stages: design, implementation, testing, and maintenance. We believe that a potent programming paradigm has to be effective along the entire software engineering process.

Behavioral Programming (BP) is a programming paradigm focused on creating complex system behavior by combining relatively simple scenarios [1]. It provides means of building systems from such scenarios. Each scenario is a simple sequential thread of execution and is thus called a *b-thread*. B-threads often resemble the textual requirements that they describe and thus promote alignment of code and requirements. The set of b-threads in a model is called a behavioral program

(*b-program*). During run-time, all b-threads participating in a b-program are combined, yielding a complex behavior that is consistent with all said b-threads. B-threads communicate using a simple event-based protocol: they can request, wait-for, and block events. A central event arbiter repeatedly selects events that are requested and not blocked. see Section 2.1 for a deeper description of BP.

The work presented here is on extensions and applications of the BP paradigm that are part of the joint efforts of our group to make it more accessible and usable for a wider set of the application. Specifically, it examines the possibility to enhance BP by means of Constraint Solvers, Reinforcement Learning and Petri Nets. We show that using the extended idioms, together with adequate analysis and execution algorithms, gives power to developers and enhances their productivity.

1.1 Contributions

This thesis makes the following contributions:

1. A stable, unified implementation of BP in Python. This implementation is used as an infrastructure for the research presented here and serves others in our research group and outside of it.

2. An experimental protocol for a more natural and abstract system modelling. In this protocol, a combination of BP and Deep Reinforcement Learning allows for giving abstract instructions to a system and for teaching it to follow them.

3. A method for allowing direct specification of liveness requirements in BP. By integrating Reinforcement Learning into the event selection mechanism, we show that liveness requirements can be enforced in execution without adding unnecessary external constraints.

4. A methodology for Discrete Event Systems (DES) modelling using BP and Petri Net. The methodology offers to model system requirements with BP prior to implementing it with petri net, to avoid premature implementation decisions.

1.2 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 discusses the foundations on which this thesis relies, and presents related work. Chapter 3 briefly introduces BPpy: a Behavioral Programming package in Python [5]. Chapter 4 details an experimental protocol for a more natural and abstract system modelling through a BP based implementation presented in [6]. Chapter 5 describes a method allowing the direct specification of liveness requirements in BP. Chapter 6 proposes a methodology for Discrete Event Systems modelling using BP and Petri Net. Chapter 7 concludes and proposes future research directions.

Chapter 2

Preliminaries and Related Work

This chapter presents the backdrop for the work reported in this thesis. We start by describing Behavioral Programming (BP) [1], the programming paradigm on which this thesis is based (Section 2.1). Next we cover some background related to BP extensions described in Chapter 3, Chapter 4 and Chapter 5 - Constraint Solvers, Reinforcement Learning and Liveness Properties of Concurrent Programs (Section 2.2, Section 2.3 and Section 2.4). Each will be covered by first providing general background and then taking a look at existing BP related work concerning it. We finish the chapter covering background and related work to Chapter 6 on Discrete Event Systems (DES) and Petri Net (Section 2.5).

2.1 Behavioral Programming

The BP paradigm focuses on constructing reactive systems incrementally from their expected behaviors [1]. When creating a system using BP, developers specify a set of scenarios. The scenarios define what the system may, must, or must not do. Each scenario is a simple sequential thread of execution and is thus called a *b-thread*. B-threads are normally aligned with system requirements, such as "go to ball" or "turn to goal".

The set of b-threads in a model is called a behavioral program (*b-program*). During run-time, all b-threads participating in a b-program are joined, yielding a complex behavior that is consistent with all said b-threads. Unlike other paradigms, BP does not force the developers to choose a single behavior for the system to use. Instead, the system is allowed to choose any compliant behavior. This enables the run-time to optimize program execution at any given moment, e.g., based on available resources. The fact that all possible system behaviors comply with the b-threads (and hence with the system requirements), assures that whichever behavior is chosen, the system as a whole will perform as specified.

In [7], Harel, Marron, and Weiss proposed a simple protocol for b-thread synchronization, as follows. The protocol consists of each b-thread submitting a statement before the selection of each

event that the b-program produces. The selection is done by an application-agnostic mechanism that takes all the statements of the b-threads into account. When a b-thread reaches a point where it is ready to submit a statement, it synchronizes with its peers. The statement declares which events the b-thread requests, which events it waits for (but does not requests), and which events it blocks (forbids from happening). After submitting the statement, the b-thread is paused. When all b-threads have submitted their statements, we say that the b-program has reached a *synchronization point*. Then, a central event arbiter picks a single event that was requested and was not blocked. Having picked an event, the arbiter resumes all b-threads that requested or waited for that event. The remainder of the b-threads remain paused, and their statements are used in the next synchronization point. The process is repeated throughout the execution of the program. A schematic view of this protocol is presented in Figure 2.1.

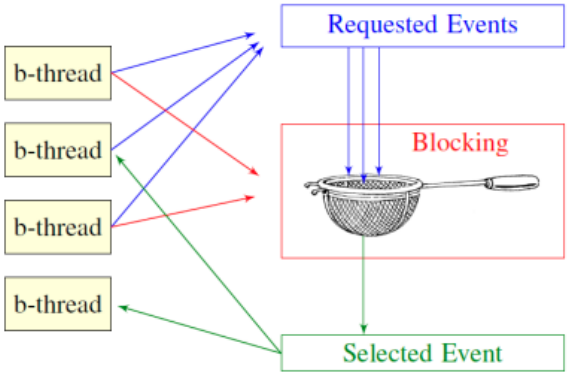


Figure 2.1: A schematic view of the execution of behavior threads

BP is implemented in a variety of languages, including Live Sequence Charts (LSC) [8, 9], JavaScript [10], Java [7], C [11] and more. Previous demonstrations of BP include a show case of a fully functional nano-satellite [12], a development tool that allows an integration of BP models with imperative code [10], a model-checking tool for verifying the correctness of BP models [13]. Research on BP cover, among others, fields discussed in this thesis and are covered in the following sections.

2.2 Constraint Solvers, SMT and Z3

Constraint solvers have become widely used and highly successful in the last decades, especially in tasks related to software analysis and verification. Largely, such methods take a set of constraints given as a formula over a set of variables as input and return a variables assignment that satisfies the formula (or state that no such variable assignment exists). Different solvers differ in the types of constraints they allow, and several popular solvers operate on constraints given in forms of first order logic.

Satisfiability Modulo Theories (SMT) Solvers are a generalization of the well-known SAT solvers, capable of handling formulas in rich fragments of first order logic. The satisfiability of the formulas is checked subject to (i.e., modulo) background theories, which intuitively restrict the search only to satisfying assignments that "make sense" according to these certain theories [14]. For example, considering the theory of arrays of integer elements with variable set $V = \{a, b\}$, the formula $\varphi_1 = (a[1] \geq b[2]) \wedge (a[3] \leq b[4])$ is satisfiable, whereas the formula $\varphi_2 = (a = b) \wedge (a[1] \neq b[1])$ is unsatisfiable. Modern SMT solvers support many theories of interest, including various arithmetic theories, the theory of uninterpreted functions, and theories of arrays, of sets, of strings, and others [15]. Further, these background theories can be combined: for instance, one can define formulas that involve arrays of integers or sets of strings, etc. The SMT problem is generally undecidable, although certain background theories afford efficient decision processes.

Z3 is a recently developed SMT solver from Microsoft Research [16]. It is aimed at solving problems that occur in software verification and software analysis. Consequently, it integrates support for a variety of theories [16]. Z3 is implemented in C++ and includes interfaces in several programming languages such as Python, Java, and Julia.

Harel et al. [14] presented an extension to the BP paradigm with dynamic generation of rich events. The extension introduces a central, application-agnostic mechanism for adding optimization to standard event selection, using Z3 SMT. It allows events with a multitude of variables and parameters, each can become an entire model [14]. It further extends the basic BP protocol with a rich set of composable constraints and functions, describing desired and undesired variable assignments, where each constraint may relate to all variables or just a subset thereof. This thesis follows concepts presented in this paper, both in implementing BPpy, as described in Chapter 3, and the controller described in Chapter 4.

2.3 Reinforcement Learning

A major challenge in developing large and complex systems is coping with the huge variety of scenarios that may arise in a rich non-deterministic environment while acting optimally within its specifications bounds. A key feature of BP is that the resulting *b-program* is often non-deterministic in the sense that it gives the machine a "freedom of choice" in situations where specifications are not complete. Since it is desirable to be able to explore executions of the models, researchers and industry have proposed tools to resolve this non-determinism by means of Synthesis, Model Checking, Planning, Priorities, and Reinforcement Learning.

Reinforcement Learning (RL) is a computational approach for understanding and for automating goal-directed learning and decision making [17]. It is differentiated from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment. RL is arguably the first field to seriously address the computational issues that arise when learning from interaction

with an environment to achieve long-term goals [17].

RL uses the formal framework of Markov Decision Processes (MDP) to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. This framework is designed to be a basic way of representing essential features of the artificial intelligence problem: a sense of cause and effect, uncertainty or non-determinism, and explicit goals.

A MDP can be expressed as a tuple $\langle S, A, R, T, \gamma \rangle$ which contains:

- A set of possible world states S.
- A set of possible agent actions A.
- A reward function $R(s, a, s')\colon S \times A \times S \to \mathbb{R}$.
- A transition function $T(s, a, s')\colon S \times A \times S' \to [0, 1]$ - the probability to move from s to s' using action a.
- A discount factor $\gamma \in [0, 1]$

In the standard RL model, an agent is connected to its environment via perception and action, as depicted in Figure 2.2. The agent can select which action to take in a given state, which has an effect on the next state it sees. However, the agent does not necessarily control the dynamics of the environment completely. The environment, upon receiving these actions, returns the next state and reward.

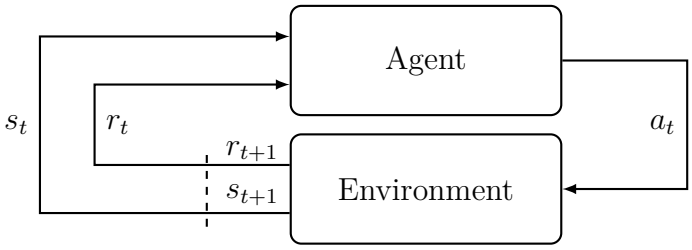


Figure 2.2: A standard RL model

RL agents learn a policy $\pi\colon S \to A$, a function that dictates the best action an agent can take in any world state in order to maximize the future cumulative discounted reward [17]:

$$R^t = R(s^t \xrightarrow{a^t} s^{t+1} \cdots) = \sum_{k=t}^{\infty} \gamma^k R(s^k, a^k, s^{k+1}).$$

In order to learn the optimal policy, RL agents make use of value functions. The action value function, $Q^\pi(s, a)$, is an estimate of the expected future reward that can be obtained from (s, a) when following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R^t | s^t = s, a^t = a].$$

The optimal action value function

$$Q^*(s,a) = max_\pi Q^\pi(s,a)$$

provides maximal values in all states and is computed by solving the Bellman equation:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma max_{a'} Q^*(s',a')\Big).$$

Q-learning [18] is a RL algorithm to learn the optimal action value function. It amounts to an incremental method for dynamic programming which imposes limited computational demands [18]. Furthermore, it works by successively improving its evaluations of the action value function in each learning iteration

$$Q^{new}(s^t,a^t) \leftarrow Q^{old}(s^t,a^t) + \alpha\Big(R(s^t,a^t,s^{t+1}) + \gamma \cdot max_a Q(s^{t+1},a) - Q^{old}(s^t,a^t)\Big)$$

where $\alpha \in (0,1]$ is the learning rate. In [18], Watkins and Dayan showed that Q-learning converges to the optimal value function $Q^*(s,a)$ with probability 1 so long as all actions are repeatedly sampled in all states and the state-action values are represented discretely. The optimal policy, π^*, is then

$$\pi^*(s) = \text{argmax}_{a \in A}\, Q^*(s,a).$$

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of RL [19]. Most successful RL applications that operate on these domains have relied on hand-crafted features integrated with linear value functions or policy representations. Clearly, the performance of such systems greatly relies on the quality of the feature representation. Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [20, 21] and speech recognition [22, 23]. These techniques exploit a variety of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines, and recurrent neural networks, and have utilized both supervised and unsupervised learning. In order for these techniques to be beneficial for RL with sensory data, reinforcement learning approaches are augmented with deep neural networks.

Deep Q-Networks (DQN) algorithm [24] approximates the value function $Q(s,a)$ with a deep neural network that outputs a set of action values $Q(s,\cdot;\theta)$ for a given state input s, where θ are the parameters of the network. There are two major components of DQN that make this work. First, it uses a separate target network that is copied every τ steps from the regular network so that the target Q-values are more stable. Second, the agent adds all of its experiences to a replay buffer $\mathcal{D}^{replay}$, which is then sampled uniformly to perform updates on the network.

The *Double Deep Q-Networks* (DDQN) algorithm [25] uses the current network to calculate the argmax over the next state values and the target network for the value of that action. The double

DQN loss is

$$J_{DQ}(Q) = (R(s,a) + \gamma Q(s_{t+1}, a_{t+1}^{max}; \theta') - Q(s,a;\theta))^2$$

where θ' are the parameters of the target network, and $a_{t+1}^{max} = argmax_a Q(s_{t+1}, a; \theta)$. Separating the value functions used for these two variables reduces the upward bias that is created with regular DQN updates.

Prioritized Experience Replay (PER) algorithm [26] modifies the DQN agent to sample more important transitions from its replay buffer more frequently. The probability of sampling a particular transition i is proportional to its priority, $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where the priority $p_i = |\delta_i| + \epsilon$, and δ_i is the last TD error calculated for this transition and ϵ is a small positive constant to ensure all transitions are sampled with some probability. To account for the change in the distribution, updates to the network are weighted with importance sampling weights, $w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta$, where N is the size of the replay buffer and β controls the amount of importance sampling with no importance sampling when $\beta = 0$ and full importance sampling when $\beta = 1$. β is annealed linearly from β_0 to 1.

In [27], Eitan and Harel extended the semantics of BP with reinforcements, allowing applications that specify, in addition to what should be done or not done at every step, also broader goals. Reinforcements are captured in [27] by b-threads, each one contributing a narrow assessment of the current situation relative to a longer-term goal. Leveraging the unique structure of b-programs, an application-agnostic reinforcement learning mechanism transforms the reinforcements into event-selection decisions which improve over time. This ability to learn and adapt allows the programmer to focus on specifying needs, while leaving the details and optimizations to the program, thus simplifying development. This thesis continues, in some ways, the work in [27], both by extending its capability and integrating deep reinforcement learning, to allow the integration of high-dimensional state or action space, as presented in Chapter 4, and by ensuring liveness properties in BP, as presented in Chapter 5.

In another, loosely related yet notable work, Weinstock [28] extended the BP execution mechanism with on-line heuristic search in program state space that allows programmers to develop programs while relying on a "smart" event selection mechanism to resolve non-determinism in a way that maximizes a defined heuristic function.

2.4 Liveness Properties of Concurrent Systems

A prerequisite for the concurrent systems analysis is a model of the system under consideration. Transition systems are often used in computer science as models to describe the behavior of such systems. An execution of a transition system can be viewed as an infinite sequence of states starting in some initial state $s_0 \in init$. Each state after s_0 results from executing a single atomic event $e \in E$ in the preceding state (For a terminating execution, an infinite sequence is obtained by

repeating the final state):

$$s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_i} s_i \ldots$$

A Linear-Time (LT) property is a set of infinite sequences of elements of events, i.e, a subset of E^ω. Since a program also defines a set of sequences of states, we say that a property holds for a program if the set of event sequences defined by the program is contained in the property.

It is useful to distinguish two classes of properties, safety and liveness, first described in [29]. Informally, a safety property stipulates that some "bad thing" doesn't happen during execution. As such, it can be violated by finite runs. Examples of safety properties include mutual exclusion, deadlock freedom, partial correctness, and first-come-first-serve [30].

Informally, a liveness property stipulates that a "good thing" happens during execution [29]. Examples of liveness properties include starvation freedom, termination, and guaranteed service [30]. In starvation freedom, which states that a process makes progress infinitely often, the "good thing" is making progress. In termination, which implies that a program does not run forever, the "good thing" is the fulfillment of the final instruction. Finally, in guaranteed service, which states that every request for service is satisfied eventually, the "good thing" is receiving a service. As such, it can only be violated by infinite runs. More formally, an LT property P is a liveness property if

$$pref(P) = \{\rho : \rho \sqsubseteq \sigma, \sigma \in P\} = (E)^*.$$

Liveness properties verification has received some attention in research on BP. As pointed by Bar-Sinai [31], a key difference between safety and liveness properties in BP is that for safety properties, the blocking idiom often allows b-programs to be correct-by-construction: if an event should not happen in a given situation, it can just be blocked. That means we can only specify safety properties in BP. On the other hand, ensuring that a b-program complies with liveness requirements requires verification.

Harel et al. [13] presented a methodology and a supporting model-checking tool for verifying behavioral programs. They defined and verified liveness properties as follows. B-threads can mark b-program states as hot. The model checker verifies liveness properties using a nested DFS algorithm, searching for cycles in the state graph that contain only hot states, . That is to say, the verified liveness property is that always eventually, the b-program is not in a hot state. Bar-Sinai [31] continued the ideas sketched in [13] and realized them by providing a tool able to detect liveness violations in b-programs [10]. He further distinguished between cycles in which a single b-thread is indefinitely hot and cycles where the b-program is indefinitely hot, but each of its b-threads is non-hot infinitely often.

The mechanism we describe in Chapter 5 allows for the direct specification of liveness requirements. Our definition of liveness in relation to BP is inspired by both [13] and [31]. To the best of our knowledge, the work presented in Chapter 5, is the first that has been made on liveness enforcing execution of b-programs. Furthermore, the work presented in Chapter 5 provides a general

mechanism for such execution, allowing infinite state space represented by real valued variables.

2.5 Discrete Event Systems and Petri Net

Discrete Event Systems (DES) are discrete-state, event-driven systems, where the discrete state changes at a discrete-time instant due to the occurrence of events. Manufacturing systems and service systems, database systems, traffic networks, integrated command, control, communication and information systems, etc., are examples of DES. Petri Net (PN) [32] is a popular modeling formalism for DES since they can provide abundant structural information about the system, and they are amenable to mathematical analysis. A PN (example depicted in 6.1) is a directed bipartite graph with *places*, *transitions*, and *arcs*, where transitions (squares) represent the discrete events, places (circles) each holding zero or more tokens, and weighted arcs. Arcs are used for connecting only places to transitions or vice versa. A transition is enabled if each of its input places holds a number of tokens that is higher or equal than its associated arc's weight (pre-conditions). PNs are non-deterministic, meaning any enabled transition may fire. Firing a transition removes the weighted amount of tokens from each of its input places and adds the weighted to each of its output places (post-conditions).

The PN formalism is valuable for analyzing DES, allowing to check system properties, such as boundedness, liveness, and diagnosability. However, as pointed out by Giua and Silva [33], while the use of PN with state specifications is a very mature area, their use in the design of systems from general behavioral specifications has not been equally successful. The latter issue can in some cases lead to incorrect specifications, faulty implementations, and inconsistent system behavior. Finding a more general approach to system modeling is still an open problem driving several developments in the PN field.

The ability to structurally define the behavior of the whole system as a function of the behavior of its subsystems is a key factor in designing systems from a general behavior description. Hence there has been much research concentrating on PN compositionality and sub-PN interactions representation. Several works introduced a compositional extension of PN using process algebras [34, 35]. They provide an approach for a high-level description of interactions, communications, and synchronizations between PNs. In another work, Baldan et al. [36] represents PN interactions by introducing open PNs, a generalization of the ordinary model. In open PNs, some places, designated as open, reflect interaction with other nets. Concretely, an open place can function as an input or an output (or both), meaning that external PN can put or remove tokens from it. While notable work has been done in this field, we argue that the BP approach addresses compositionality more naturally, with the ability to compose behaviors without direct consideration of mutual dependencies.

An important part of generalizing PN modeling is the ability to represent an interface of the system with the environment. Plain PNs are not adequate to model systems that can interact

with their environment or, in another view, are only partially specified. The above-mentioned open PNs [36] can also model external interaction, where some nets in the whole model represent the system's environment. In another related work, reactive PNs [37] are addressing this issue by defining reactive semantics to PN: Splitting the set of transitions to internal and external and modifying its firing rules. The modified rules state that if an external transition is enabled it may fire, while in contrast, internal transition, when enabled, must fire. Such behavior is desired in systems that are specified to react as a consequence of external events. We view the ability to accurately model real-time scenarios in reactive systems to be of great importance. BP approaches an external environment interaction in its semantics [38] and implementations [10] using the mechanism of super-steps that capture the priority of external events over external ones that reflect the notion of logical execution time [39].

Cardozo et al. [40] discusses issues related to software engineering and composability of PNs introducing a context-oriented programming approach. While the focus of this paper is not DES, we believe that the issues raise by Cardozo et al. are relevant to DES and PN modeling in general. Their approach provides a mechanism to compose a specification of separated PN models that interact via a protocol that allows them to inhibit one another and react to each other. Specifically, each context is represented in a specific PN structure, applying concepts from several PN extensions [37, 41, 42]. This structure manages the activation and deactivation of the context in response to stimuli from the environment. They show that such mechanisms are useful for analyzing context-based specifications. We believe that this is a key property for allowing alignment of a specification with its requirements, as demonstrated in Chapter 6. In our setting, the components of the model are directly cued from paragraphs in the specification.

Chapter 3

BPpy: Behavioral Programming in Python

In this chapter, we briefly present BPpy: a Behavioral Programming package in Python [5]. This package was developed to facilitate this thesis research and its implementations as will be discussed in the following chapters. BPpy was released for open access and is being used in several types of research in the field of Behavioral Programming (BP) [6, 43]. We start this chapter by providing the motivation and goals that drove the development of BPpy (Section 3.1). We then introduce it, its design, and how it can be executed using a tutorial example (Section 3.2). We finish the chapter by presenting a Z3 solver integration implemented in BPpy (Section 3.3).

3.1 Motivation and Goals

Python has become one of the most popular programming languages recently. Its design philosophy emphasizes code simplicity and readability [44]. The simple core concepts and knowledge required to write a Python code can get one quite far. It is therefore only natural that the BP paradigm, which is focused on simplicity of programming [1], will be implemented in Python. BPpy is based on basic Python idioms, preserving its simplicity and readability.

While most programming languages have mechanisms to write and reuse libraries of code, Python is particularly blessed in having a large and extensive standard library built into the core language, along with a thriving ecosystem of third-party modules. This ecosystem includes a wide data analysis and Machine Learning (ML) support. Numerous stable frameworks, such as numpy [45], tensorflow [46], baselines [47] and z3-solver [16] are implemented and supported in Python. This enables to research and integrate BP with different data analysis and ML concepts and algorithms easily using BPpy.

As mentioned, BPpy was originally developed to facilitate this thesis research and its implementations. It was only later that, due to demand, the package was released and used in several types

of research in the field of BP. Implementing and releasing a reliable BP platform in Python serves as a common research infrastructure, mainly in the field of data analysis and ML integration. BPpy promotes collaboration and code reuse, therefore accelerating BP research and adoption aside from its traditional community.

3.2 Introducing BPpy

To better describe the package's core concepts, we now turn to a tutorial example of a simple b-program, written in BPpy. The example, presented in Listing 3.1, is an adaptation of one of the sample b-programs presented in [7], the HOT/COLD example. The described system has the following requirements:

1. When the system loads, do 'HOT' three times.
2. When the system loads, do 'COLD' three times.
3. Two actions of the same type cannot be executed consecutively.

```
from bppy import *

@b_thread
def add_hot():
    yield {request: BEvent("HOT")}
    yield {request: BEvent("HOT")}
    yield {request: BEvent("HOT")}

@b_thread
def add_cold():
    yield {request: BEvent("COLD")}
    yield {request: BEvent("COLD")}
    yield {request: BEvent("COLD")}

@b_thread
def control_temp():
    e = BEvent("Dummy")
    while True:
        e = yield {waitFor: All(), block: e}
```

Listing 3.1: HOT/COLD example BPpy implemention

Following an early implementation presented in [14], each b-thread is implemented as a Python generator. A generator is a function that can pause itself and pass data back to its caller at any point, using the `yield` idiom (the equivalent of the `bp.sync` in the BPjs library [10]). It can then be subsequently resumed when it is re-invoked with the language's `next` idiom. The BPpy infrastructure mimics the parallel execution, as follows. It calls each generator sequentially, waiting for it to yield a sync statement, in the form of a Python dictionary containing events or event sets

labeled by the keys `request`, `block`, `waitFor`. When all statements are collected, a central event arbiter selects a single event that was requested and was not blocked. Having selected an event, the arbiter resumes all b-threads that requested or waited for that event.

```
b_program = BProgram(bthreads=[add_hot(), add_cold(), control_temp()],
                     event_selection_strategy=SimpleEventSelectionStrategy(),
                     listener=PrintBProgramRunnerListener())
b_program.run()
```

Listing 3.2: Instantiating and executing b-programs in BPpy

Listing 3.2 demonstrates how b-programs are instantiated and executed in BPpy. The `BProgram` instance is composed of a list of b-threads (Listing 3.1 for instance), an event selection strategy (arbiter) and an optional b-program listeners. The default event selection mechanism, implemented in the class `SimpleEventSelectionStrategy`, selects an event uniformly from the set of events that were requested and not blocked. Optionally, if a more complex mechanism is required, an alternative event selection strategy can be used by implementing a class that inherits the abstract class `EventSelectionStrategy`. A listener is notified when the b-program starts, ends and events are selected, allowing to embed a b-program in a host application. It can be implemented by inheriting the abstract class `BProgramRunnerListener`.

3.3 Z3 Solver Integration

In many reactive systems, controllers often involve multiple numeric and discrete parameters. While some BP frameworks [10] allows for attaching rich data, including numerical values, to the events, its event selection is discrete. For systems that require rich controllers with requirements that pose complex relations between the variables in the events, this may cause a complication. To cope with this complication the integration of solver-based event selection protocol in BP has been a subject of considerable recent works [48, 14].

BPpy implements Z3 satisfiability modulo theory (SMT) solver integration following the concepts presented in [14]. The implementation is based on z3-solver [16] Python package. To demonstrate this integration, the HOT/COLD example SMT variation is presented in Listing 3.3. In this extension, events are represented as the set of SMT variables (e.g. hot cold Boolean variables). In each synchronization round, the b-threads specify requested, blocked and waited-for constraints over the variables in the form of logical or arithmetic statements to be satisfied. After collecting all constraints, the new execution mechanism invokes the Z3 SMT solver to find an assignment to the variables embedded in the events. The assignment is then returned by the `yield` command of the b-thread.

```
from bppy import *

hot = Bool('hot')
cold = Bool('cold')

@b_thread
def three_hot():
    for i in range(3):
        yield {request: hot , waitFor: hot}

@b_thread
def three_cold():
    for j in range(3):
        yield {request: cold , waitFor: cold}

@b_thread
def control_temp():
    m = yield {}
    while True:
        if is_true(m[cold]):
            m = yield {block: cold}
        if is_true(m[hot]):
            m = yield {block: hot}

@b_thread
def mutual_exclusion():
    yield {block: And(cold ,hot), waitFor: false}
```

Listing 3.3: HOT/COLD SMT example BPpy implemention

While it is the only BP implementation the integrates a constraint solver to the event selection protocol, BPpy is missing tools for b-program analysis and verification (implemented in the BPjs library [10] for example). This is left for future work, both by realizing some of the concepts presented in the following chapters and new research directions.

Chapter 4

The RoboCup Use Case: Integrating Solvers and Deep Reinforcement Learning

As systems are expected to be more sophisticated and autonomous, developers require tools that allow them to effectively develop software while avoiding over-specification. This chapter describes a BP based implementation of a controller for a player in a simplified RoboCup-type game and is a part of a paper published in MDETOOLS'19 workshop on model-driven engineering tools [6].[1] Towards substantiating the observation that it is easier to explain things to an intelligent agent than to a dumb compiler, we demonstrate how the combination of Behavioral Programming (BP) and Deep Reinforcement Learning (DRL) allows for giving abstract instructions to the robot and for teaching it to follow them after a short training session.

The code that we show is a simple proof-of-concept model that controls the robot to get a hold of the ball, take it to the goal, and try to score a goal. We use this example to explain some variants of the modelling approach that we are developing and to describe some guidelines of how and when each of them can be used. The complete code for the controllers presented in this chapter can be found in `https://github.com/bThink-BGU/Papers-2019-MDETools`.

The first model is presented in Section 4.2. We show how a constraints solver can be used to provide a richer coordination protocol that allows modellers to design modules with higher cohesion and lower coupling. Specifically, we demonstrate how the solver mechanism allows the components of the model to use a rich constraint language to represent aspects of the behavior that involve arithmetic and logical specifications. This, in turn, allows for a better alignment with such requirements that we usually observe in design documents of robotic systems.

The second model is presented in Section 4.3. There, we give an example of how instructions

[1] Most of this chapter is taken word-by-word from [6]. I was involved in all the writing of this paper and focused mostly on Section 4.3.

can be simplified when assuming a certain level of intelligence of the mechanism that runs the code. In the same way that it is easier to train an animal to carry a complex mission in a robust way than it is to program a computer to achieve the same level of robustness, we show that it is easier to program a machine when we know that it can interpret our commands intelligently than it is to give instructions to a dumb compiler. Specifically, we show that a combination of BP and DRL allows for specifying behavior patterns that the robot can learn to use at the right times. We demonstrate how we achieved a visible level of simplification in the model even after a short training session.

The rest of the chapter continues as follows. In Section 4.1 we provide, for completeness, a short description of the RoboCup game. The controllers are then described in Section 4.2 and Section 4.3 as said above. Finally, we discuss the two implementations and future work.

4.1 The Simplified RoboCup-Type Game

The organizers of the MDETOOLS'19 workshop on model-driven engineering tools created a simulation of a simplified RoboCup-type game inspired by `https://ssim.robocup.org`. In this workshop participants were challenged to use modelling tools in the context of a controller for a virtual robot, as described below. The description here was adopted from the text that appears on the website of the workshop at `https://mdetools.github.io/mdetools19/challengeproblem.html`.

The simulation, as shown in Figure 4.1, contains a ball and two players red (player2) and blue (player1) who compete against each other. By default, the blue player can be controlled by the keyboard or by the simulation where its movements are random. The challenge was to create a model to control the red player. The objective of each player is to shoot as many goals as possible.

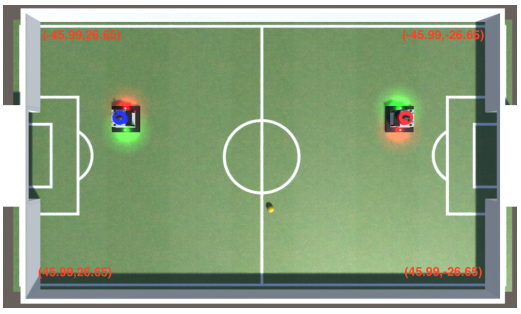


Figure 4.1: The RoboCup-Type Game

At the beginning of the simulation, the ball lies in the field's kickoff position and both players are placed equidistant from the ball on their respective sides. The game lasts for a fixed amount of time. Upon acquiring the ball, the player has a fixed time period during which they can possess the ball (time period resets after losing possession). If the possession time period expires for the player holding, the ball is ejected from the player, and the player is not allowed to move for a certain amount of time. After a goal, the simulation is reset to the initial state.

4.2 Using a Solver for Real Valued Variables

In this section, we describe a simple implementation of a controller for the red robot, using the BPpy Z3 integration described in Section 3.3. In robotic systems, such as the one at hand here, the commands for controlling the robots, often involve numerical values, usually including vectors with several dimensions, such as velocity, spin, and a combination of complex actuation commands. Hence, for an efficient controller implementation, a solver-based event selection protocol is required.

```
@b_thread
def invariants():
    yield {block:
             Not(And(forward >= -MAX_PWR,
                     forward <= MAX_PWR,
                     Or(spin == 0,
                        spin == MAX_SPIN,
                        spin == -MAX_SPIN))),
           waitFor: false}
```

Listing 4.1: The `invariants` b-thread for the solver based controller

In Listing 4.1, the `forward` variable models that the speed in which the robot drives itself forward, must be in a certain range, and that the variable `spin` must get one of the three values given in the constraint. Since the b-thread specifies that it does not wait for anything, these constraints will be held as invariants. Note that we had to add the `waitFor: false` here, since the default is `waitFor: true`, i.e., a b-thread is awakened after each event unless something else is specified.

```
@b_thread
def move_towards_ball():
    m = yield {}
    while True:
        dst = dist_ball_robot(m)
        if dst > TOO_CLOSE:
            if dst < TOO_FAR:
                m = yield {request: forward == grad(dst)}
            else:
                m = yield {request: forward == MAX_PWR}
        else:
            if dst > (2 * TOO_CLOSE - TOO_FAR):
                m = yield {request: forward == grad(dst)}
            else:
                m = yield {request: forward == -MAX_PWR}
```

Listing 4.2: The `move_towards_ball` b-thread for the solver based controller. The first `yield` waits for the first state of the game (event) and assigns it to the variable `m`. The b-thread then computes the distance between the robot and the ball using `m` and continues by requesting an assignment to `forward`, based on this data.

The rest of the controller logic is also implemented by b-threads that submit constraints. For example, the `move_towards_ball` b-thread presented in Listing 4.2, takes care of assignments to the variable `forward`. In this simple example, all the constraints are equalities, meaning that this b-thread requests specific actions, showing that it is possible to directly express ordinary requests also in the solver-based version BP.

The extended semantics allows also to request infinite sets, even non-enumerable ones, as demonstrates by the `spin_to_ball` b-thread depicted in Listing 4.3. This b-thread specifies whether it wants the spin to be positive, negative, or zero (i.e., it requests an infinite set of values to the `spin` variable). Since these constraints are submitted to the solver, along with all the other constraints (including the invariants given in Listing 4.1), the value that the solver assigns to the `spin` variable is uniquely determined in our case. In other cases, when the specification has multiple solutions, the solver chooses one arbitrarily. For example, in a time in a game where the distance between the ball and the robot is larger than `TOO_FAR` and the robot is facing the ball, the execution of these b-threads will yield the event `{forward:MAX_PWR, spin:0, ...}` that will be assigned to the variable `m` (i.e., the solver assignment). The main takeaway here is that the solver-based mechanism allows for breaking the specification into modules in new ways, allowing for a better alignment of the modules with the aspects of the behavior that the designers and the users perceive. It also allows for more effective modelling of systems that require a focus on numerical and Boolean fields in the events.

```
@b_thread
def spin_to_ball():
    m = yield {}
    while True:
        if is_ball_in_robot(m):
            ang = angle_between_robot_and_ball(m)
            if ang > MAX_ANG:
                m = yield {request: spin > 0,
                           block:spin <= 0}
            elif ang < -MAX_ANG:
                m = yield {request: spin < 0,
                           block:spin>=0}
            else:
                m = yield {request: spin == 0}
        else:
            m = yield {}
```

Listing 4.3: The `spin_to_ball` b-thread for the solver based controller. Using the solver based semantics allows the b-thread to only specify the direction of the spin, leaving the specification of the exact value to other b-threads.

We also added b-threads to handle the suction variable. See `https://github.com/bThink-BGU/Papers-2019-MDETools`.

4.3 Using Deep Reinforcement Learning for Simplicity and Robustness

In this section, we present an attempt to make the instructions described in Section 4.2 even more natural and abstract. A motivating example can be found in the `move_towards_ball` b-thread. As described in Listing 4.2, this b-thread takes care of assignments to the variable `forward`. In order to do so, it uses the predefined constants — `TOO_CLOSE` and `TOO_FAR`. In the original code, we calculated these constants using a manual trial-and-error, while here, we use Reinforcement Learning (RL) for this task. This setting allows the instructions to be simpler, more robust, and easier to maintain if the simulation changes. We show that with a combination of a rich solver and deep neural networks, the approach can also be used with multidimensional events and actions that contain numerical fields.

RL is a computational approach for understanding and for automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment [17]. In the standard RL model, an agent is connected to its environment via perception and action, as depicted in Figure 2.2.

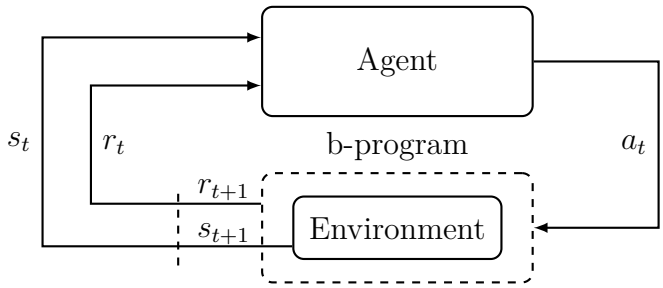


Figure 4.2: A modified behavioral RL model

In our setting, depicted in Figure 4.2, the agent interacts with the b-program, which encapsulates the environment, in our case — the RoboCup game. The current state of the environment, s_t, consists of the player's position, compass, suction, and the ball's position. The action chosen by the agent, a_t, is used by the b-threads to modify the robot's behavior. In this challenge, we targeted the task of grabbing the ball. As shown in Listing 4.4, the reward in each step, r_t, is defined by the `get_ball_reward` b-thread. This general model allows high modelling flexibility and generality, by allowing a model to be applied to various RL algorithms with different parameters of the environment.

```
@b_thread
def get_ball_reward():
    m = yield {block: reward != -0.01}
    while True:
        if is_ball_in_robot(m):
            m = yield {block: reward != 1}
        else:
            m = yield {block: reward != -0.01}
```

Listing 4.4: The `get_ball_reward` b-thread. The agent's reward in each step is defined by `is_ball_in_robot` auxiliary function.

In order to assist with the task of ball grabbing, the b-threads that are related to this task were simplified, by removing the exact conditions and parameters for the different actions. Instead, the activation of the actions now depends on the agent's commands. For example, in the modified `move_towards_ball` presented in Listing 4.5, the following parameters for the `forward` variable are controlled by the agent: `half_speed_forward`, `full_speed_forward`, `half_speed_backwards`, and `full_speed_backwards`.

```
@b_thread
def move_towards_ball():
    m = yield {}
    while True:
        if not is_ball_in_robot(m):
            if half_speed_forward:
                m = yield {request: forward == MAX_PWR/2}
            if full_speed_forward:
                m = yield {request: forward == MAX_PWR}
            if half_speed_backwards:
                m = yield {request: forward == -MAX_PWR/2}
            if full_speed_backwards:
                m = yield {request: forward == -MAX_PWR}
        else:
            m = yield {}
```

Listing 4.5: The modified `move_towards_ball` b-thread. Assignments to `forward` variable are dictated by the RL agent actions: `half_speed_forward`, `full_speed_forward`, `half_speed_backwards`, `full_speed_backwards`.

The `spin_to_ball` b-thread, which controls the direction of the spin, was modified in the same manner, as shown in Listing 4.6. Note that our goal is not to provide an interface to RL, but rather to provide programmers with a natural programming and modelling tool that applies RL under the hood. The key takeaway here is that we demonstrate how an intelligent execution mechanism can interpret more abstract commands that allow programmers to better break their models into modules that are aligned with the behavioral aspects that they perceive.

```
@b_thread
def spin_to_ball():
    m = yield {}
    while True:
        if not is_ball_in_robot(m):
            ang = angle_between_robot_and_ball(m)
            if need_to_spin and ang > 0:
                m = yield {request: spin > 0}
            elif need_to_spin and ang < 0:
                m = yield {request: spin < 0}
            else:
                m = yield {request: spin == 0}
        else:
            m = yield {}
```

Listing 4.6: The modified `spin_to_ball` b-thread. The `spin` variable is controlled by the `need_to_spin` agent action.

To cope with the high-dimensional sensory inputs of the simplified RoboCup-type simulation state, our implementation, uses a *Deep Q Network* (DQN) [24] implementation of [49]. The simulation state is being fed into the network as input. The output action of the DQN is then used by the `move_towards_ball` and `spin_to_ball` b-threads to modify the game controller. Note that we are not just using DRL to achieve an automatic generation of a control strategy, our goal in this work is to simplify the software-engineering practices for robots software design. The end result of the example we have experimented with, is that the programmer could only specify modes (e.g. `half_speed_forward`, `full_speed_forward`, `half_speed_backwards`, and `full_speed_backwards`) and have the execution engine decide automatically when to activate each of them (based on a training session). Notice that this example shows how a constraint solver, which allows rich events in BP, and DRL, which allows learning from rich data, can complement each other.

4.4 Discussion and Future Work

In this chapter, we have demonstrated a BP approach to program a robot in the RoboCup game. We started by demonstrating how behavior modeling can be achieved with b-threads that coordinate through a more sophisticated protocol that applies a constraint solver to choose the actions. Specifically, we argued that, for robotic systems where the actuation commands are multidimensional, it is easier to let the b-threads specify equalities, inequalities, and logical composition thereof, than propose lists of events. We then concluded by showing how Deep Reinforcement Learning protocols can facilitate further simplification of the model, by allowing programmers to specify abstract behavior and have an automatic tool sort the implementation details.

The goal of this chapter was to present how the RoboCup challenge can be tackled using BP, towards comparing it with other tools that will be presented at the conference. In future work, we hope to be able to compare our approaches with other tools, learn how to improve our tools, and provide feedback to others. While both methods have the potential to significantly simplify the models, the latter seems to be more promising. Nonetheless, additional work is required to better formalize it and demonstrate its contribution.

Chapter 5

Live Execution of Behavioral Programs using Reinforcement Learning

The design and development of complex systems is a research area with many challenges and is a subject of considerable work in recent years. A major challenge in developing such systems is ensuring that it complies with different liveness requirements. In this chapter, we propose to give developers the ability to directly specify liveness requirements easily and intuitively using Behavioral Programming (BP). By integrating Reinforcement Learning (RL) into the event selection mechanism, we show that specified liveness requirements can be enforced in b-program execution.

Liveness properties, as defined by Lamport [29], require infinite runs to conform to a requirement, and do not constrain finite runs. Accordingly, no finite witness can be used as proof for a violation of liveness requirement, while a violation of a given safety requirement admits a finite witness. In the context of BP, as pointed by Bar-Sinai [31], a key difference between safety and liveness properties is that for safety properties, the blocking idiom often allows b-programs to be correct-by-construction: if an event should not happen in a given situation, it can just be blocked. That means we can only specify safety properties in BP. On the other hand, ensuring that a b-program complies with liveness requirements requires additional verification.

We claim that by allowing the direct specification of liveness properties using an extension to BP semantics can enhance programming while remaining simple and intuitive for software developers. Using RL, the program is given the freedom to learn to comply with liveness requirements and adapt according to its experience from the system's environment. Such adaptivity is highly desirable, as the programmer cannot always anticipate how the environment will react. Also, it allows the developer to focus on specifying needs, while leaving the details to the program, thus emphasizing the advantages of behavioral programming, and offering a natural and modular way to program. By utilizing the system experience and integrating it in the b-program event selection mechanism,

we allow the alignment of liveness requirements during execution.

We used Q-learning [18] as a learning mechanism for our suggested framework. While it may be inefficient in some domains, it is generally accepted that Q-learning (and RL in general) offers generality in the sense that it can be applied to a range of problems. For example, Q value function approximation can be used to learn directly from high-dimensional state space [24]. With that said, our framework is flexible, allowing the integration of other algorithms solving Markov Decision Processes (MDP), e.g., searching or planning algorithms.

The rest of this chapter is organized as follows. In Section 5.1 we describe the motivation for the concepts described in this chapter. Section 5.2 presents our method for liveness requirements specification and execution in BP. In Section 5.3 we provide a formal proof of the method correctness and demonstrate it through examples in Section 5.4. We finish with a short discussion in Section 5.5.

5.1 Motivation

The motivation for the concepts described in this chapter can be found in an example of a simple b-program, written using BPpy. This example is an adaptation of the HOT/COLD example, presented in Section 3.2. Consider a system with the following requirements:

1. When system loads, do action 'A' at least three times.
2. When system loads, do action 'B' at least three times and then do action 'I' forever.
3. At least one action 'B' must be executed between two actions 'A'.

Listing 5.1 shows a b-program (a set of b-threads) that attempts to fulfill these requirements. It consists of three b-threads, each responsible for fulfilling one of the above-mentioned requirements.

```
@b_thread
def req_1():
    for i in range(3):
        yield {request: BEvent("A")}
@b_thread
def req_2():
    for i in range(3):
        yield {request: BEvent("B")}
    while True:
        yield {request: BEvent("I")}
@b_thread
def req_3():
    while True:
        yield {waitFor: BEvent("A")}
        yield {waitFor: BEvent("B"), block: BEvent("A")}
```

Listing 5.1: The HOT/COLD example adaptation. The b-program specifies to do action 'A' and 'B' three times each (followed by infinite 'I' actions). It further dictates that at least one action 'B' must be executed between two actions 'A'.

A careless review of the program's structure might be deceiving and lead to the belief it is aligned with the system requirements. However, in examining the program's possible runs, while all desired runs are possible ($ABBABAIII\ldots$, $ABABABIII\ldots$, etc.), other undesired runs ($BBBAIII\ldots$, $ABBBAIII\ldots$, etc.) are permitted. Such runs are undesired since the requirements state that the system will do action 'A' *at least* three times. This issue stems from a limitation in BP semantics - it cannot specify liveness properties. This presents a problem with BP, where the inability to directly specify a certain type of requirements ("The system will eventually do 'A' three times") might lead to faulty implementations or inconsistent system behavior.

Notice that, we can avoid the undesired run in the above example by dictating exactly when each action should be performed, either by implementing a single-threaded program or by additional b-thread. However, in complex systems, doing so is a hard task prone to over-specification. Hence, we claim that by allowing the direct specification of liveness properties using an extension to the BP semantics can enhance programming while remaining simple and intuitive for software developers.

We suggest the *must finish* idiom to specify liveness requirements. The example in Listing 5.2 demonstrates a b-thread specifying that "The system will eventually do 'A' three times". Using RL, the program is given the freedom to learn to comply with liveness requirements and adapt according to its experience from the system's environment. By utilizing the system experience and integrating it in the b-program event selection mechanism, we allow the alignment of liveness requirements during execution.

```
@b_thread
def req_1():
    for i in range(3):
        yield {request: BEvent("A"), must_finish: True}
```

Listing 5.2: A b-thread specifying that "The system will eventually do 'A' three times" using the `must_finish` idiom

5.2 RL Based Live Execution Mechanism

We now present our suggested mechanism for liveness enforcing execution. The definition below follow [7] and were modified to include the notion of a state *must finish* labeling. B-threads and their collective execution definition is based on labeled transition systems. Recall that a labeled transition system is defined as a quadruple $\langle S, E, \rightarrow, init\rangle$, 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 [50]. 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 5.2.1. (b-thread). A b-thread is a tuple $\langle S, E, \rightarrow, init, B, R, L\rangle$ where $\langle S, E, \rightarrow, init\rangle$ forms a labeled transition system, $R : S \rightarrow 2^E$ associates a state with the set of events requested by the b-thread when in it, $B : S \rightarrow 2^E$ associates a state with the set of events blocked by the b-thread when in it, and $L : S \rightarrow \{0, 1\}$ is a labeling function that indicates if the state is to be finished(*must finish*).

Definition 5.2.2. (b-program). A b-program is a set of b-threads $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i, L_i\rangle\}_{i=1}^n$

In line with the definitions of b-threads in [7], an abstraction is chosen in which a state of a b-thread is defined only at the synchronization points. Specifically, we do not explore intermediate transitions in the code that b-threads execute between exiting one point and entering the next. This abstraction provides for reducing the size of the state-space, based on the assumption that b-threads interact only through events, which means that, for the purpose of liveness properties, their activities between synchronization points are considered atomic and are independent of each other. We accord with Harel et al. [13], noting that this assumption is reasonable, and does not unduly constrain the capabilities of behavioral programs.

Definition 5.2.3. (run of a b-program). A run of a b-program, $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i, L_i\rangle\}_{i=1}^n$, is a run of the labeled transition system $\langle S, E, \rightarrow, init\rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = init_1 \times \cdots \times init_n$, 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, there may be more than one run of a set of b-threads, depending on the order in which events are selected from the set of requested and unblocked events [7].

Definition 5.2.4. (live b-program run). We define a run of a b-program, $\{\langle S_i, E_i, \rightarrow_i, init_i, L_i, R_i, B_i\rangle\}_{i=1}^n$, a live run if for each b-thread i and time t, there exists $t' > t$ such that

$$L_i(s_i^{t'}) = 0$$

Simply stating, we define a live run if each b-thread enters a state not labeled as *must finish* infinitely often.

Following [31], we distinguish between a case in which all b-threads are not labeled as *must finish* infinitely often, and a case where all b-threads are not labeled as *must finish* simultaneously infinitely often. These cases can represent different types of liveness violations. Note that, while we chose to focus on the first type in this chapter, the proposed method can be further generalized to fit the second.

Definition 5.2.5. (b-program i-liveness MDP). A b-program i-liveness MDP is a b-program, with an extension of a reward function R_i, which rewards live runs for a b-thread i based on it's *must finish* labeling function L_i:

$$R_i(s, e, s') = \begin{cases} -1 & L_i(s_i) = 0 \wedge L_i(s'_i) = 1 \\ 1 & L_i(s_i) = 1 \wedge L_i(s'_i) = 0 \\ 0 & else \end{cases}$$

Note that such MDP is defined for each b-thread i and differs only by its reward function R_i. This setting allows us to distinguish between live runs of b-threads based on the entire b-program dynamics, which affects the b-thread by the request and block idioms.

For clarity, in this chapter, we refer to actions as events and consequently note e instead of a (or E instead of A).

Definition 5.2.6. (Q-compatible run). Given $Q_i \colon S \times E \to \mathbb{R}$, the action value function, estimating the expected future reward R_i, we define a run $s^0 \xrightarrow{e^0} s^1 \xrightarrow{e^1} \cdots$ of a set of b-threads to be Q_i-compatible run if for each time t

$$\sum_{t'=0}^{t-1} R_i(s^{t'}, e^{t'}, s^{t'+1}) + Q_i(s^t, e^t) > -1$$

We define a run of a set of b-threads to be Q-compatible run if it is Q_i-compatible for each b-thread i.

Definition 5.2.7. (sampling distribution for Q-compatible runs). We consider the following sampling distribution over the Q-compatible runs - for each state s^t choose event e^t uniformly from all events such that for each b-thread i,

$$\sum_{t'=0}^{t-1} R_i(s^{t'}, e^{t'}, s^{t'+1}) + Q_i(s^t, e^t) > -1.$$

We suggest the following mechanism for liveness enforcing execution of BP. Given a b-program with the extended *must finish* label, as defined in Definition 5.2.2, a Q-learning [18] algorithm will learn the optimal action value function Q^*. We then use the sampling distribution described in Definition 5.2.7 for Q^* as the b-program event selection strategy. In the following sections, we show a formal proof of correctness and demonstrate the suggested mechanism through examples.

5.3 Formal Proof of Correctness

In this section we show that using the optimal value function Q^*, we can enforce liveness requirements in our b-program execution. We do so by proving that a live b-program run is Q^*-compatible (Theorem 5.3.1) and that a Q^*-compatible b-program run is almost surely live (Theorem 5.3.2). Our proof assumes all b-thread's initial states are not labeled as *must finish*. We believe that this assumption is reasonable, and does not restrict the capabilities of our suggested framework.

Claim 5.3.0.1. *For every infinite b-program run $l = s^0 \xrightarrow{e^0} s^1 \xrightarrow{e^1} \cdots$, every b-thread i, and time $t \geq 0$:*

$$\sum_{k=0}^{t} R_i(s^{k-1}, e^{k-1}, s^k) = \begin{cases} 0 & \text{if } L_i(s_i^t) = 0; \\ -1 & \text{if } L_i(s_i^t) = 1. \end{cases}$$

Proof. We will prove the claim by induction on t. The base case is given by the assumption that $L_i(s_i^0) = 0$ in every run. Assuming that the claim is true for $t-1$, If $L_i(s_i^{t-1}) = L_i(s_i^t)$ then $R(s^{t-1}, e^{t-1}, s^t) = 0$ and the claim follows. If $L_i(s_i^{t-1}) = 0$ and $L_i(s_i^t) = 1$ then $R(s^{t-1}, e^{t-1}, s^t) = -1$ and we get that

$$\sum_{k=0}^{t} R_i(s^{k-1}, e^{k-1}, s^k) = \sum_{k=0}^{t-1} R_i(s^{k-1}, e^{k-1}, s^k) - 1 = -1.$$

If $L_i(s_i^{t-1}) = 1$ and $L_i(s_i^t) = 0$ then $R(s^{t-1}, e^{t-1}, s^t) = 1$ and we get that

$$\sum_{k=0}^{t} R_i(s^{k-1}, e^{k-1}, s^k) = \sum_{k=0}^{t-1} R_i(s^{k-1}, e^{k-1}, s^k) + 1 = 0.$$

We get that all cases are consistent with the claimed equation. □

Claim 5.3.0.2. *For every infinite b-program run $l = s^0 \xrightarrow{e^0} s^1 \xrightarrow{e^1} \cdots$ and b-thread i, let $(t_k)_{k=0}^n$ be the sequence of times where $R_i(s^{t_k}, e^{t_k}, s^{t_k+1}) \neq 0$. The length of the sequence can be finite or infinite or empty, i.e., $n \in \mathbb{N} \cup \{\infty, -1\}$. Then for every $0 \leq k \leq n$:*

$$R_i(s^{t_k}, e^{t_k}, s^{t_k+1}) = (-1)^{k+1}.$$

Proof. Since $L_i(s_i^0) = 0$, it is clear from Definition 5.2.5 that $R_i(s^{t_0}, e^{t_0}, s^{t_0+1}) = -1$. Assuming towards contradiction that there is a $k \geq 0$ such that

$$R_i(s^{t_k}, e^{t_k}, s^{t_k+1}) = R_i(s^{t_{k+1}}, e^{t_{k+1}}, s^{t_{k+1}+1}).$$

Since $R_i(s^t, e^t, s^{t+1}) = 0$ for every $t_k < t < t_{k+1}$, by the same definition, $L_i(s^{t_k+1}) = L_i(s^{t_{k+1}})$. This contradicts the definition of R_i where it is apparent that $L_i(s^{t_k+1}) = L_i(s^{t_{k+1}})$ implies

$$R_i(s^{t_k}, e^{t_k}, s^{t_k+1}) \neq R_i(s^{t_{k+1}}, e^{t_{k+1}}, s^{t_{k+1}+1}).$$

□

Claim 5.3.0.3. *For every infinite live b-program run $l = s^0 \xrightarrow{e^0} s^1 \xrightarrow{e^1} \cdots$, b-thread i, time $t \geq 0$, and $\gamma < 1$:*

$$\sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) > \begin{cases} -1 & \text{if } L_i(s_i^t) = 0; \\ 0 & \text{if } L_i(s_i^t) = 1. \end{cases}$$

Proof. Similarly to the sequence used in Claim 5.3.0.2, let $(q_k)_{k=0}^{n_q}$ be the sequence of times after t where $R_i(s^{q_k}, e^{q_k}, s^{q_k+1}) = 1$, and $(r_k)_{k=0}^{n_r}$ be the sequence of times after t where $R_i(s^{r_k}, e^{r_k}, s^{r_k+1}) = -1$. Note that, since run l is live we have that $n_q \geq n_r$, otherwise the run ends with infinitely many must-finish states. If both sequences are empty, since $L_i(s_i^0) = 0$, it is clear from Definition 5.2.5 that $L_i(s_i^t) = 0$. In this case, all the rewards are zero and the claim holds trivially. Furthermore, if $L_i(s_i^t) = 1$ and $n_r < 0$ then $(q_k)_{k=0}^{n_q}$ is not empty, i.e., $n_q \geq 0$ or else the run ends with infinitely many must-finish states. In this case we get that

$$\sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) = \sum_{k=0}^{n_q} \gamma^{q_k} > 0.$$

If the sequences are not empty and $L_i(s_i^t) = 1$, from Claim 5.3.0.2 we get that $q_k < r_k$ for each $k \leq n_r$ and then

$$\sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) \geq \sum_{k=0}^{n_r} (\gamma^{q_k} - \gamma^{r_k}) > 0.$$

If the sequences are not empty and $L_i(s_i^t) = 0$, from Claim 5.3.0.2 we get that $r_k < q_k < r_{k+1}$ for each $k < n_r - 1$ and

$$\sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) \geq -\gamma^{r_0} + \sum_{k=0}^{n_r-1} (\gamma^{q_k} - \gamma^{r_{k+1}}) > -1.$$

We get that the claim holds in all cases. □

Claim 5.3.0.4. *If $Q_i^*(s^t, e^t) > 0$ then there is a path $s^t \xrightarrow{e^t} s^{t+1} \xrightarrow{e^{t+1}} \cdots \xrightarrow{e^{t+m_t-1}} s^{t+m_t}$ such that $L(s^{t+m_t}) = 0$.*

Proof. Using the optimal policy π^*, we construct a path by defining, for every $t' > t$, $a^{t'} = \pi^*(s^{t'})$ and choosing $s^{t'+1}$ to be the only state such that $T(s^{t'}, a^{t'}, s^{t'+1}) = 1$. There is only one such state since the b-program transitions, as defined in Definition 5.2.3, are deterministic. Assume, towards contradiction, that $L(s^{t'}) = 1$ for every $t' \geq t$. Then

$$Q_i^*(s^t, e^t) = \sum_{t'=t}^{\infty} \gamma^{t'} R_i(s^{t'}, e^{t'}, s^{t'+1}) = 0$$

which contradicts the assumption. This gives us that the path that we have constructed is as required. □

Theorem 5.3.1. *A live b-program run is Q^*-compatible.*

Proof. Let $l = s^0 \xrightarrow{e^0} s^1 \xrightarrow{e^1} \cdots$ be a live run. To show that l is Q^*-compatible we now show it is Q_i^*-compatible for each b-thread i, which means showing the term in Definition 5.2.6 holds for every time t. If $L_i(s_i^t) = 0$, from Claim 5.3.0.1 we get that

$$\sum_{k=0}^{t} R_i(s^k, e^k, s^{k+1}) = 0$$

and, as shown in Claim 5.3.0.3

$$\sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) > -1.$$

If $L_i(s_i^t) = 1$, from Claim 5.3.0.1 we get that

$$\sum_{k=0}^{t} R_i(s^k, e^k, s^{k+1}) = -1$$

and, as shown in Claim 5.3.0.3

$$\sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) > 0$$

In both cases, when adding the terms together we get that for every time t

$$\sum_{k=0}^{t} R_i(s^k, e^k, s^{k+1}) + \sum_{k=t}^{\infty} \gamma^k R_i(s^k, e^k, s^{k+1}) > -1.$$

By the definition of the optimal value function of b-thread i, Q_i^*

$$\sum_{k=0}^{t} R_i(s^k, e^k, s^{k+1}) + Q_i^*(s^t, e^t) > -1.$$

We get that run l is Q_i^*-compatible for each b-thread i and is therefore Q^*-compatible. □

Theorem 5.3.2. *A Q^*-compatible b-program run is almost surely live.*

Proof. Let π be the policy defined in 5.2.7 using the optimal value function Q^*. We will show that π will generate a live run with probability one. Since, by definition, π always draws a Q^* compatible runs, we have

$$\sum_{k=0}^{t-1} R_i(s^k, e^k, s^{k+1}) + Q_i^*(s^t, e^t) > -1$$

for all $t \geq 0$. To generate a non-live run, π needs from some point of time, t_0, to always visit states

that are labeled as *must finish*, i.e., $L_i(s_i^t) = 1$ for all $t > t_0$. By 5.3.0.1, for all $t > t_0$,

$$\sum_{k=0}^{t-1} R_i(s^k, e^k, s^{k+1}) = -1$$

and we get that

$$Q_i^*(s^t, e^t) > 0.$$

Therefore, by 5.3.0.4 for every $t > t_0$ there is a path $s^t \xrightarrow{\hat{e}^t} \hat{s}^{t+1} \xrightarrow{\hat{e}^{t+1}} \cdots \xrightarrow{\hat{e}^{t+m_t-1}} \hat{s}^{t+m_t}$ such that $L(\hat{s}^{t+m_t}) = 0$. Assume that this is the first such index, i.e., $L(\hat{s}^{t+m_t-1}) = 1$ and we get that

$$\sum_{k=t}^{t'-1} R_i(\hat{s}^k, \hat{e}^k, \hat{s}^{k+1}) = 0$$

for every $t \leq t' < t + m_t - 1$. This means that all the states along this path satisfy

$$\sum_{k=0}^{t-1} R_i(s^k, e^k, s^{k+1}) + \sum_{k=t}^{t'-1} R_i(\hat{s}^k, \hat{e}^k, \hat{s}^{k+1}) + Q_i^*(\hat{s}^{t'}, \hat{e}^{t'}) > -1$$

and that π could have chosen this path. The probability that it will not choose any of these paths is zero. □

5.4 Examples

With all concepts in place, we now present two examples of behavioral programs and demonstrate the effectiveness of our suggested method. The examples were implemented and tested on BPpy [5], and are available in `https://github.com/tomyaacov/BPLivenessRL.git`.

5.4.1 Sokoban

The first example is Sokoban (depicted in Figure 5.1), a classic transportation puzzle game, where the player has to push a number of boxes onto given target locations. Because boxes can only be pushed (as opposed to pulled), many moves are irreversible, and mistakes can render the puzzle unsolvable. We dub such scenarios "live locks" (example depicted in Figure 5.2). Using this game characteristic we demonstrate how a Sokoban playing b-program is constructed with liveness requirements, specifying every box should be pushed to a target location eventually.

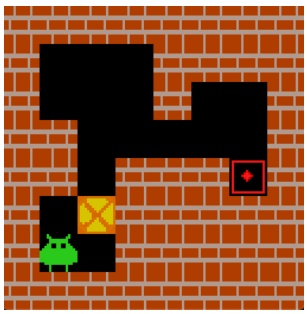


Figure 5.1: A Sokoban game example (graphics taken from [51]).

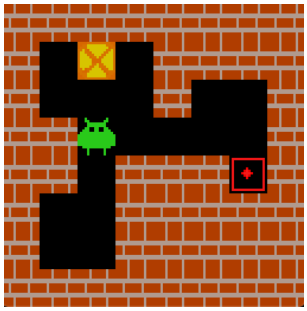


Figure 5.2: A "live lock" example in Sokoban (graphics taken from [51]). This lock occurs when a player cannot push a box to a target, making the puzzle unsolvable.

Sokoban has been in the focus of the combinatorial search community ever since the seminal work of Junghanns and Schaeffer [52], who devised advanced heuristics and pruning methods for this domain. It was also demonstrated in several works in the field of RL [53, 54]. In this chapter, we are not interested in solving Sokoban optimally, but to demonstrate how a b-program can, using the suggested method, enforce liveness requirements and avoid entering "live locks" in execution. Note that the Sokoban b-program is not the main use case that we intend for our tools, it is mainly presented for demonstration purposes and for testing how far our algorithm can scale.

In our b-program, the player is modeled in a single b-thread (Listing 5.3). The b-thread requests to move in all four directions invariably. Once an event is selected, the location of the player is changed using the `event_to_new_location` function. Note that events contain the location of the player, which will be used by other b-threads.

```
@b_thread
def player(i, j):
    directions = ["Up", "Down", "Left", "Right"]
    while True:
        e = yield {request: [BEvent(d, {"i": i, "j": j}) for d in directions]}
        i, j = event_to_new_location(e)
```

Listing 5.3: The Sokoban player b-thread. The b-thread requests to move in all four directions invariably

In Listing 5.4 we model each wall object as a b-thread. The b-thread blocks all events of moving to the wall location (computed using the `get_blocked_events` function). Since the b-thread specifies that it does not wait for anything (`EmptyEventSet()`), these blocked events are held as invariants.

```
@b_thread
def wall(k):
    global walls_list
    i, j = walls_list[k]
    block_list = get_blocked_events(i, j)
    yield {block: block_list, waitFor: EmptyEventSet()}
```

Listing 5.4: A Sokoban single wall b-thread. The b-thread blocks all events moving to the wall's location.

Each box object is also modeled as a b-thread, as listed in Listing 5.5. The b-thread blocks all events of moving the box to a wall or other boxes using the `double_obj_move_set` event set. Once an event is selected, the b-thread moves the box if required. We model the liveness property we require, specifying that every box should be eventually pushed to a target location, using the `must_finish` idiom.

```
@b_thread
def box(k):
  global box_list, walls_list, target_list
  i, j = box_list[k]
  while True:
    neighbors_list = find_adjacent_objects(box_list[k], walls_list + box_list)
    double_obj_move_set = EventSet(block_action(neighbors_list))
    box_in_tar = box_list[k] in target_list
    e = yield {block:double_obj_move_set,waitFor:All(),must_finish:not box_in_tar}
    new_player_location = event_to_new_location(e)
    if new_player_location == box_list[k]:
      new_box_location = event_to_2_steps_trajectory(e)
      box_list[k] = new_box_location
```

Listing 5.5: A Sokoban single box b-thread. The liveness property we require, specifying every box should be pushed to a target location eventually is modelled in this b-thread using `must_finish` idiom.

For evaluation, we have tested the standard Q-learning [18] algorithm as a learning mechanism for our suggested method. The tested b-programs represent rooms in several configurations as depicted in Figure 5.3. Our results show that Q values in all b-programs reached the optimal values after a short training session. Consequently, our mechanism avoided live locks in all evaluated runs.

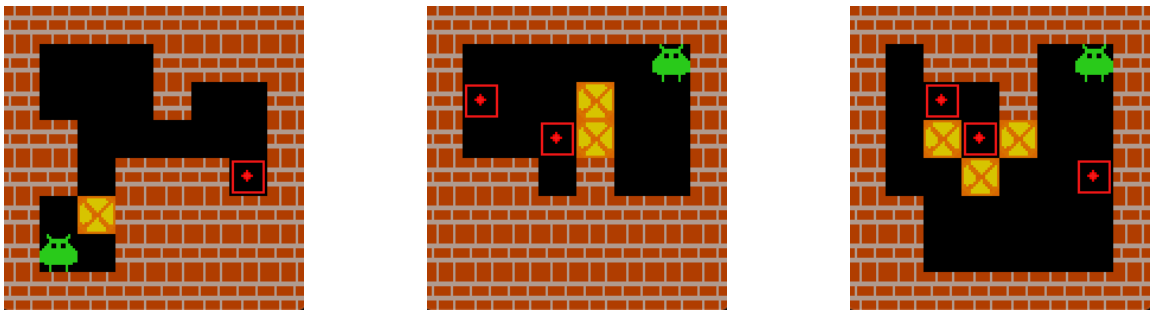


Figure 5.3: Tested Sokoban environments (Graphics taken from [51]).

5.4.2 Single Lane Bridge

The theoretical part of this work, as explained in Section 5.2, assumes that the run of the b-program is infinite. While the above b-program is indeed infinite, the Sokoban game is finite as the game may run into a "live lock" or end when all boxes are in place. While it is a useful example for studying how our method is capable of handling such systems, we now complement this example and demonstrate how our method has the potential to synthesize behaviors in robotic systems such as autonomous cars that need to solve different road situations [55].

The second example is an adaptation of a well-known problem in concurrent systems analysis, the Single Lane Bridge problem [56]. In the problem, depicted in Figure 5.4, a car is attempting to cross a bridge over a river from one direction while n cars are attempting to cross it from the other. The bridge is wide enough only to permit a single lane of traffic. Consequently, cars can only pass the bridge if they are moving in the same direction. For clarity, we will refer to the car moving from left to right as the red car and the cars moving from right to left as the blue cars.

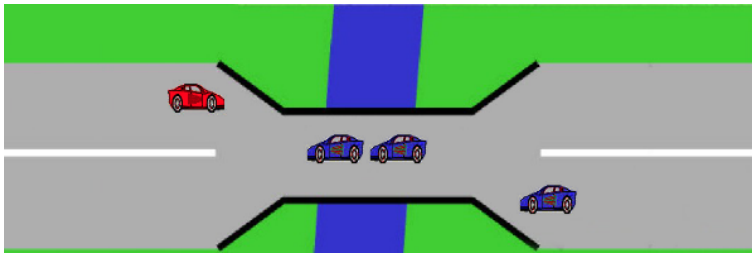


Figure 5.4: The Single Lane Bridge Problem (Graphics taken from [56]).

In cases where cars from both sides enter the bridge simultaneously a "live lock" occurs (Figure 5.5), in which both cars cannot advance while the simulation resumes. Both lanes are circular, meaning that cars from both directions can potentially (if not stuck) cross the bridge infinitely often. The liveness property we require is that the red car will cross the bridge infinitely often.

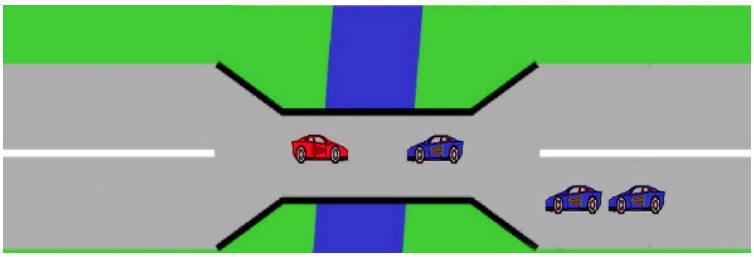


Figure 5.5: A 'live lock" example in the Single Lane Bridge problem. This lock occurs when cars from both sides enter the bridge simultaneously (Graphics taken from [56]).

In the constructed b-program, the red car is modeled in two separate b-threads as listed in Listing 5.6. The first b-thread continuously requests to either `"Wait"` or `"Move"` and advance the red car if required. The second b-thread forbids the red car from entering the bridge in case that a blue car is already on the bridge by blocking the `"Move"` event. Note that this case differs from the above-mentioned "live lock" where both cars enter the bridge simultaneously.

```
@b_thread
def red_car():
    e = yield {request: [BEvent("Wait"), BEvent("Move")]}
    advance_red_car(e)

@b_thread
def control_red_crossing():
    while True:
        if any_blue_on_bridge() and red_attempting_to_cross():
            yield {block: BEvent("Move"), waitFor: All()}
        else:
            yield {waitFor: All()}
```

Listing 5.6: The two b-threads modeling the red car dynamics

Each blue car is modeled in a separate b-thread as shown in Listing 5.7. At initialization, The blue car location is sampled. During execution, all blue cars advance at a constant speed.

```
@b_thread
def blue_car(i):
    sample_blue_car_location(i)
    while True:
        yield {waitFor: All()}
        advance_blue_car(i)
```

Listing 5.7: A single blue car b-thread

The liveness property we require, that red car will cross the bridge infinitely often, is modeled in a simple b-thread depicted in Listing 5.8. While waiting for the red car to cross the bridge the b-thread state is labeled as `must_finish` (computed using the `red_passed` function).

```
@b_thread
def red_pass_bridge_infinitly_often():
    while True:
        yield {waitFor: All(), must_finish: not red_passed()}
```

Listing 5.8: A b-thread specifying that the red car will cross the bridge infinitely often

To evaluate our suggested method, we've executed and tested the effectiveness of several successful RL algorithms. The b-program state space, which includes cars location, contains real valued variables. Coping with such input type requires a function approximation for the action value function Q. Our experiment included *Deep Q-Networks* (DQN) [24], *Double Deep Q-Networks* (DDQN) [25] and the *Prioritized Experience Replay* (PER) [26] algorithms implemented in [49]. The evaluation was made on b-programs containing one, two, and three blue cars.

Table 5.1 summarizes the results of our experiment. The metric we chose for evaluation measures the "successful" runs rate. A "successful" run is a run in which a "live lock" did not occur. As a baseline, we calculated the metric for the standard random execution of the b-program, sampling from all requested and non-blocked events uniformly. The results show that, in the case of a single blue car, all algorithms reached a perfect "successful" runs rate after a relatively short training session (10^5 events). However, in the case of two and three blue cars, the algorithms did not yield a perfectly "live lock" safe application. We attribute this to the inaccuracy of Q values approximations reached in the algorithms. Nonetheless, the DQN algorithm appears to be the most stable and applicable for this example. It resulted in a nearly perfect "live lock" safe application in the case of two blue cars and achieved the highest "successful" runs rate with three blue cars.

Cars	**Events**	**Random**	**DQN**	**DDQN**	**PER**
1	10^5	67.8%	100%	100%	100%
2	10^6	54.6%	99.4%	86.2%	91.4%
3	10^7	42.4%	87.7%	82.1%	82.9%

Table 5.1: Results summary - "successful" runs rate. Evaluation was made on b-programs containing one, two and three blue cars after learning over the listed amount events.

5.5 Discussion and Future Work

In this chapter, we have demonstrated a method allowing the direct specification of liveness requirements in BP easily and intuitively. By integrating RL into the event selection mechanism, we showed that the specified liveness requirements can be enforced without adding unnecessary external constraints. We demonstrated how this extension can enhance the programming experience and how it is simple and intuitive for software developers. Besides using the method for executing complete applications, it can be used during development for early identification of liveness

conflicts. The BP approach is particularly suitable for such incremental refinement, as it supports refinement by the addition of new b-threads without changing existing ones.

Our setting considers a case in which liveness requirements are defined in each b-thread separately. This is one of the two types of Liveness specifications proposed by Bar-Sinai [31]. While we chose to focus on this type, the proposed method can be further generalized to fit the other type (where liveness properties relate to the state of the b-program, not only of a single b-thread). This is left for future studies.

The mathematical theory shows that given a solution to the optimal Q values, we can form a liveness enforcing event selection strategy. This, however, is still not fully demonstrated in the experimental section since approximations that we have obtained for some solutions were not accurate enough. Specifically, some examples in Section 5.4.2 did not yield a perfectly "live lock" safe application. It is generally accepted that RL offers generality in the sense that it can be applied to a range of problems but may be inefficient in some cases. In our case, further research is needed to establish a more reliable learning algorithm for our method that offers the same level of generality.

Chapter 6

An Improved Modeling Methodology for Discrete Event Systems

Discrete Event Systems (DES) are discrete-state, event-driven systems, where the discrete state changes at a discrete-time instant due to the occurrence of events. Manufacturing systems and service systems, database systems, traffic networks, integrated command, control, communication and information systems, etc., are examples of DES. Petri Net (PN) [32] is a popular modeling formalism for DES since they can provide abundant structural information about the system, and they are amenable to mathematical analysis.

In this chapter, we argue that the state-of-the-art PN formalism for modeling DES is driving modelers to premature design decisions that lead to incomplete and inaccurate models that are hard to generalize and maintain. We will support our claim with a known DES benchmark, called the *level-crossing benchmark*, and demonstrate the inaccuracy of PN models of this benchmark. To address this problem, we propose to model the system in two phases: (1) Model the requirements with the Behavioral Programming (BP) paradigm, allowing for a direct specification, execution, and verification of the requirements; and (2) Implement the requirements with PN. As we will show, the additional phase that we propose has the following advantages:

- It supports a modular specification approach where each module isolates a specific aspect of the system behavior.
- It allows for verifying different properties of the requirements, such as reachability, liveness, boundedness, diagnosability, etc.
- The verification algorithm can be executed in a compositional way, thus handling large search spaces.
- It separates the requirements from the implementation, thus helping modelers to avoid premature implementation decisions.

- It supports the current PN-based practices for DES.
- The transition to PN is automatic - we give transnational semantics from the BP model to the PN, and vice versa.

Furthermore, to avoid the necessity of verifying both the BP model and the PN model, we propose an algorithm and a tool for comparing the two models and testing their equivalency. Thus, our approach provides a way for DES modelers to verify the correctness of the requirements and verify that their PN implementation is aligned with the requirements.

The rest of the chapter continues as follows. In Section 6.1 we give a general description of the level-crossing benchmark. We model the benchmark requirements with BP in Section 6.2 and present an implementation with PN in Section 6.3. In Section 6.4 we provide an algorithm for comparing the two models and use it to indicate the premature implementation decision of the benchmark modelers and the err of their PN model. To "ease" the transition for DES modelers, in Section 6.5 we provide translational semantics between the two models. We conclude the chapter with a short discussion (Section 6.6).

6.1 The Level-Crossing Benchmark

The level-crossing (LC) domain was first presented by [57] and modeled with PN. It was later used in various research areas of PN modeling and software safety analysis [58, 59, 60]. Although the original model was modified in some of these works to pertain features to the relevant study, they all followed the same general behavior of [57].

Mazzeo et al. [60] presented a systematic approach for a PN specification for the LC domain. They defined the model as a controller for a gate at a railway crossing — an intersection between a railway line and a road at the same level. The railway line has a sensor that signals the controller whenever the train is approaching, entering the crossing zone, and departing. Based on the senors' signals, the barrier raises and lowers the gate, ensuring the safety of the trains, i.e., that a train cannot be in the crossing zone while the barrier is up.

While the system behavior is not explicitly specified as a set of requirements, we have extracted the following requirements as we understand them, and we will later refine them:

R1. The railway sensor system dictates the exact event order: train approaching, entering and then leaving. Also, there is no overlapping between successive train passages.

R2. The barriers are lowered when a train is approaching and then raised as soon as possible.

R3. A train may not enter while barriers are up.

R4. The barrier may not be raised while a train is in the intersection zone. The intersection zone is the area between the approaching sensor and the leaving sensor.

At system initialization, there is no train at the intersection zone and the barrier is up. Hence, a train approaching sensor will be triggered first, and the barrier will be lowered first.

Ghazel and Liu [59] extended the LC model to support multiple railways. This extension has been used by many as a benchmark for this domain [61, 62, 63] and we use it in the remaining of the chapter as a baseline for our approach.

6.2 Modeling the Requirements with BP

To emphasize the agility of BP models, we begin with a specification that handles only one railway, and we will later extend this model to support multiple railways. Following the principles of BP described in Section 2.1, each b-thread in Listing 6.1 is aligned to a single requirement of the system.

The first requirement, describing the order of the sensor's events, is specified in the first b-thread. It continuously requests to "approach", "enter" and "leave", dictating this specific order. We note that a new cycle can be launched only if the previous train has left the intersection zone, which is aligned with the requirement of no overlapping between successive train passages.

The second b-thread specifies the second requirement of the barriers behavior. The b-thread waits for a train to approach and then requests to lower the barriers. When the barriers are down it requests to raise it as soon as possible. We note that the two barrier events, Lower and Raise, can only happen consecutively, aligned with the system behavior description.

R3 is specified by the third b-thread, which blocks the train from entering while the barrier is raised. The first synchronization point (line 19) blocks the train from entering before lowering the barrier. The second synchronization point ensures that if the barrier was raised again before a train entered, then the behavior returns to its initial state, blocking the train from entering as long as the barrier is up.

Finally, the last b-thread specifies R4, blocking the raising of the barrier while there is a train in the intersection zone.

This BP model demonstrates some merits in its approach to modeling. The system was modeled in an incremental and modular manner, where each module is aligned with a single requirement and is unaware of other b-threads. The resulting modules are readable and comprehensible to all stakeholders.

```
@b_thread
def R1():
    while True:
        yield {request: Approaching()}
        yield {request: Entering()}
        yield {request: Leaving()}

@b_thread
def R2():
    while True:
        yield {waitFor: Approaching()}
        yield {request: Lower()}
        yield {request: Raise()}

@b_thread
def R3():
    while True:
        yield {waitFor: Lower(), block: Entering()}
        yield {waitFor: Raise()}

@b_thread
def R4():
    while True:
        yield {waitFor: Approaching()}
        yield {waitFor: Leaving(), block: Raise()}
```

Listing 6.1: A b-program that specifies the requirements for a single railway. Each b-thread is aligned with a single requirement. All b-threads are interweaved at run-time, yielding a complex behavior consistent with all requirements.

6.3 Modeling the System with PN

We now turn to present the PN model of the LC benchmark, as described in [59]. The model is composed of three types of subsystems: railway traffic, barrier, and barrier controller. To comply with the specified behavior of the entire system, these subsystems are combined using a set of transitions and places. As we discuss below, this combination changes the behavior of the model, causing unpredictable side effects.

The railway-traffic subsystem (depicted in Figure 6.1) specifies the dynamics of the railway using three places and three transitions corresponding to the defined train's events: approaching, entering, and leaving. The index i denotes the index of the railway, though for now, we will have only one (thus, $i = 1$).

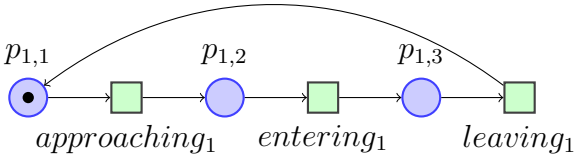


Figure 6.1: The PN LC model of the railway traffic subsystem [59].

The barrier subsystem (depicted in Figure 6.2), has two states — "up" and "down" (marked by p_7 and p_8 respectively), The barrier passively responds to the commands issued by its controller that we now describe.

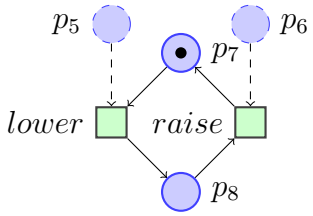


Figure 6.2: The PN LC model of the barrier subsystem [59].

The barrier-controller subsystem (depicted in Figure 6.3) provides an interface between the railway-traffic and the barrier subsystems. When a train approaches, a *closing request* is being fired and an *opening request* is fired when a train leaves. Note that this subsystem contains 2 interlocks, p_2 and p_3, which together make sure that *closing request* and *opening request* fire alternatively. In practice, this means that the barrier may be closed only if it is open and vice versa.

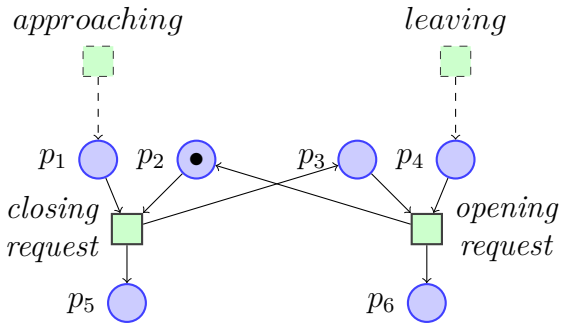


Figure 6.3: The PN LC model of the barrier-controller subsystem [59].

Finally, to address R3 and forbid the train entrance while the barrier is up, the unified model that integrates the three subsystems (depicted in Figure 6.4) includes additional interlocking state (p_9) and arcs ($lower \rightarrow p_9$, and $p_9 \rightarrow entering$).

We note that the two controller events —"closing request" and "opening request" — are not mentioned in the requirements and they are part of the model only for ensuring the safety of the system. Yet, as we formally present in the following section, the BP model also enforces the safety of the system without the use of additional events. We dub them thus — "helper events" since they are not essential for specifying the system behavior and they are only derived from a specific implementation perspective. Furthermore, we show that the helper events and the interlocking mechanism lead to undesired behaviors of the system.

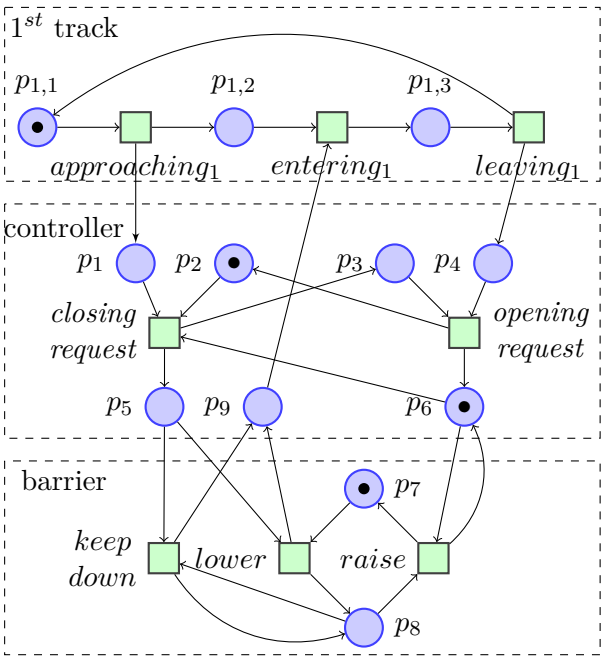


Figure 6.4: The unified PN LC model, including the three subsystems and the interlocking mechanism [59].

6.4 A Comparison of the Approaches and the Models

We begin with a definition of equivalence. In general, two models are equivalent if they yield the same set of runs, i.e., the same sequences of events. There is a complication in our case, however, since the PN model requires helper events that are not part of the BP model. For example, the following two traces are equivalent in terms of system behavior, though they have different events:

$$\begin{aligned} &BP\colon \mathit{Approaching}, && \mathit{Lower}, \mathit{Entering}, \mathit{Leaving}, && \mathit{Raise} \\ &PN\colon \mathit{Approaching}, \mathit{ClosingReq.}, \mathit{Lower}, \mathit{Entering}, \mathit{Leaving}, \mathit{OpeningReq.}, \mathit{Raise} \end{aligned}$$

As the example shows and as noted before, these events are used for synchronizing the barrier and the railway events and they are not required for comparing the resulted behavior of the two models. Thus, our equivalency definition ignores these events. For completeness, we allow helper events on either side of the comparison.

Definition 6.4.1. Models M_1 and M_2 over the event sets E_1 and E_2, respectively, are equivalent if and only if

$$\{\pi_{E_1 \cap E_2}(t) \colon t \in L(M_1)\} = \{\pi_{E_1 \cap E_2}(t) \colon t \in L(M_2)\}$$

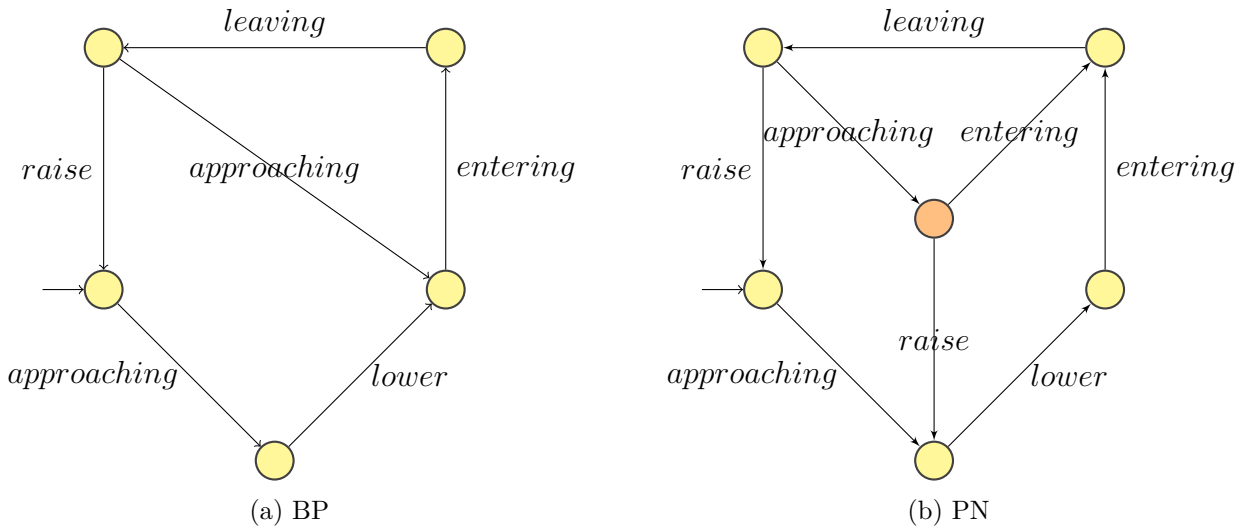
where $L(M_i)$ is a set of sequences of events, called traces, that model M_i generates, and $\pi_{E_1 \cap E_2}(t)$ is an operator that removes from a trace t all the events that are not in $E_1 \cap E_2$:

$$\pi_{E_1 \cap E_2}(t) = \begin{cases} t[0]\pi_{E_1 \cap E_2}(t[1..]) & \text{if } t[0] \in E_1 \cap E_2, \\ \pi_{E_1 \cap E_2}(t[1..]) & \text{otherwise;} \end{cases}$$

To allow traces with finite length, we also define that $\pi_E(\varepsilon) = \varepsilon$ for any E. The sequences in the sets $L(M_i)$ can be of finite or of infinite length.

Using this definition, we denote M_{BP} and M_{PN} as the BP model and the PN model (respectively). Since the trains may infinitely approach, enter, and leave the crossing zone, we use Büchi automata to represent the languages $L(M_{BP})$ and $L(M_{PN})$. A Büchi automaton contains a set of states and a transition function, where some states are defined as accepting and some as initial. The automaton accepts input if and only if there is a run of the automaton over this input that begins at an initial state and at least one of the infinitely often occurring states is an accepting state.

We generate the automata using a depth-first search (depicted in Figure 6.5). The transitions in these automata correspond to the events, and the states of these automata correspond to the synchronization points of the BP model or the reachable markings of the PN model. The accepting words of these automata represent the set of possible traces that each model may generate.



(a) BP (b) PN

Figure 6.5: The generated automaton of each of the models for a single track

To understand the significance of the difference between the two models, we analyze them using GOAL [64], a graphical tool for manipulating Büchi automata and temporal formulas. Our findings show that the resulting language for the BP model is contained in the resulting PN model — $L_{M_{BP}} \subset L_{M_{PN}}$, meaning that some runs are only possible in the PN model. One example for a word (or a trace) that is accepted only by the PN model is:

$$Approaching \cdot (Lower \cdot Entering \cdot Leaving \cdot Approaching \cdot Raise)^{\omega}$$

In this case, there are two trains on the same railway, where the second train approaches the intersection zone immediately after the first train leaves, while the barrier is down. According to this trace, even though a train is already approaching the barrier, the latter may be raised only to be lowered again immediately after. These redundant barrier actions are not aligned with the system requirements and do not stand to reason. Furthermore, as we show in Section 6.4.2, Ghazel and Lui [59] also observed this behavior and addressed it, though only for the case of multiple tracks. We believe that such behavior is derived from implementation decisions rather than specification. While the BP paradigm allows for a direct specification of the requirements as separate modules and automatic composition of them, the PN modeling approach requires the modeler to explicitly specify how the different modules interact.

6.4.1 Adjusting our Model

Despite the fact that we find the above behavior redundant, we now adjust our model to meet this behavior, starting with a redefinition of item R2:

"The barrier should be lowered after the first train approaches. Then, whenever a train is leaving, we ask to raise the barrier unless another train is approaching. When a train is approaching in this stage, it is allowed to raise and lower the barrier, unless the train is already entering."

Granted, this is a strange requirement, but this is the best we could think of given the observed behavior. We think that this was not the author's intention, only an artifact of the modelling technique used in [59].

That said, this adjustment requires the modification of the second b-thread of Listing 6.1. The original b-thread and its modified version are presented in Listing 6.2. Once a train is leaving, the controller both requests to raise the barrier, and waits for another approach. If a train approaches before the barrier is raised, then the controller again requests to raise the barrier, unless again, a train has passed. Given the new requirement, we compare its generate automata to the original automata. This time we found that the adjusted model is equivalent to the PN model.

```
@b_thread
def R2():
    while True:
        yield {waitFor: Approaching()}
        yield {request: Lower()}
        yield {request: Raise()}
```

(a) The original b-thread.

```
@b_thread
def R2_modified():
    yield {waitFor: Approaching()}
    yield {request: Lower()}
    while True:
        yield {waitFor: Leaving()}
        e = yield({request: Raise(),
                   waitFor: Approaching()})
        if e == Raise():
            yield {waitFor: Approaching()}
        else:
            e = yield {request: Raise(),
                       waitFor: Entering()}
        if e != Entering():
            yield {request: Lower()}
```

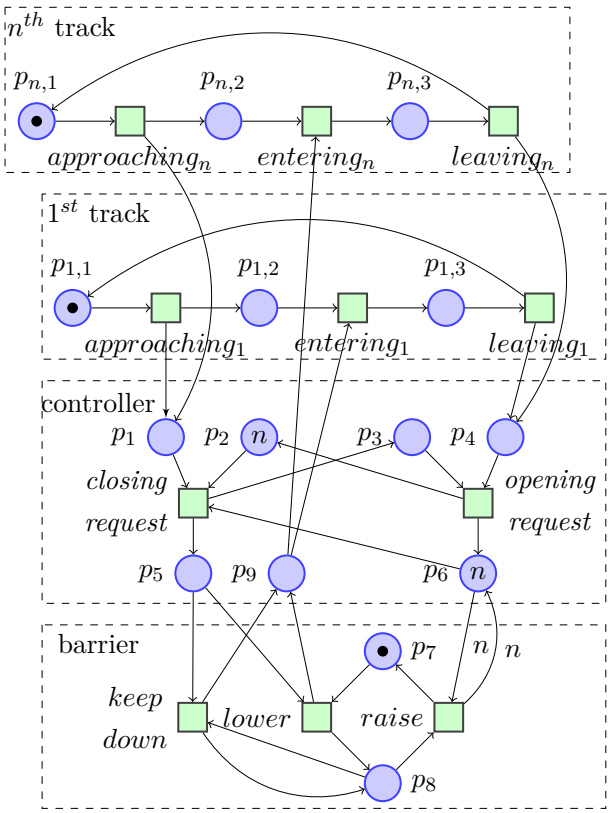
(b) The modified b-thread.

Listing 6.2: Adapting the second b-thread to the change in the requirement

6.4.2 Expanding to Multi-Track

We continue with our comparison, this time against the multiple-railway track version of the level-crossing domain. Figure 6.6 presents the extended PN model of [59] and our extended BP model.

Extending the BP model required only multiplying the b-threads by the number of tracks. Railway-specific events (i.e. Approaching, Entering, and Leaving) were assigned with an indexed while the barrier events remained the same. Since the behavior of each b-thread is valid for both the single version of the system and the multi-track version, we needed no further adaptations to the extended model.



(a)

```
@b_thread
def R1(i):
    while True:
        yield {request: Approaching(i)}
        yield {request: Entering(i)}
        yield {request: Leaving(i)}

@b_thread
def R3(i):
    while True:
        yield {waitFor: Lower(),
               block: Entering(i)}
        yield {waitFor: Raise()}

@b_thread
def R4(i):
    while True:
        yield {waitFor: Approaching(i)}
        yield {waitFor: Leaving(i),
               block: Raise()}
```

(b)

Figure 6.6: Multi-track extensions of the two models. The PN extension consists of additional arcs, tokens, and events, aside of multiplying the railway traffic sub-system [59]. The BP extension only multiplied the b-threads, maintaining the original semantics of each sub-system.

The extended PN model of [59] modified the single-track version by multiplying the railway traffic subsystem and changing the other two sub-systems as following: a "keep down" transition was added with its related arcs, two additional arcs were added ($p_6 \rightarrow$ *closing request* and *raise* $\rightarrow p_6$), tokens were added, and some arc weights were changed. Notably, this extension significantly changed the model and its semantics.

Applying the comparison algorithm of Section 6.3 shows that the extended PN model is, again, not aligned with the system requirements and is not equivalent to our model. It dictates behavior patterns that do not match the intended behavior. For instance, the barrier can be raised if a train has approached in one of the tracks. Beyond that, these modifications are not natural in our mind and increase the model's complexity, both in terms of state space and in the human ability to perceive it. We think that this small example shows the main advantage of separating the modeling of the requirement from the phase of building the model of the system. Specifically, the example shows that a live and bounded model can be designed without having to generate modeling artifacts that are not directly related to the specification.

6.4.3 Adding Faults

Fault diagnosis is of paramount importance in DES modeling and has become an active research area in recent years. The research activity in this area is driven by the needs of many different error-prone domains. When modeling, faults are often added over an existing model describing the standard system behavior. This may lead to inconsistent system behavior that is misaligned with prior requirements. This section demonstrates how our suggested method can assist modelers in verifying and analyzing the impact of faults on the initial requirements.

In the PN model detailed in [59], two classes of faults were added for diagnosis purposes, and are denoted by red-colored transitions in Figure 6.7. The first one simulates a train-sensing defect and indicates that the train may enter the level crossing zone before the barriers are lowered. The second failure indicates a defect of the barriers that result in a premature raising. It is worth noting that in order to satisfy the original requirements several additional modifications have been made. For instance, arcs $p_9 \rightarrow leaving_i$ and $entering_i \rightarrow p_9$ were added to ensure a train cannot leave if the barriers are up. This scenario can happen only if a fault enter transition is triggered, otherwise, the arc $p_9 \rightarrow entering_i$, ensuring that a train cannot enter if the barriers are up, would have been sufficient. This example, together with the extension discussed in Section 6.4.2, demonstrates a drawback in PN modeling: adding or removing new behaviors, such as faults, often requires additional adjustments to satisfy the original requirements.

The faults mentioned above can be added to our BP model in two separate b-threads as presented in Listing 6.3. The first class of faults, simulating a fault train entrance can be modeled in n separate b-threads, each responsible for a different railway i. The b-thread waits for a train on railway i to approach. It then requests a signed entering event, representing the above-mentioned fault. The second b-thread, models the second class of faults, simulating a fault barrier raise. The b-thread waits for the barrier to be lowered. It then requests the barriers to be raised, using a signed event, representing this fault. Note that, the fault events in our BP model are similar to the original non faulted events for both entering and raise respectively, and are differentiated by adding a "flag" to the event data. This setting allows existing b-threads to be affected by (by requesting and waiting) the added fault events without modifying them. Additionally, the "flag" allows the model to differentiate fault events for diagnosis purposes.

The addition of faults demonstrates the dynamical and incremental development style of BP. The described b-program is modular, in the sense that new requirements can be flexibly added as new b-threads. Since modeling often begins without faults, we believe that the ability to implement them without modifying (or even accessing) the existing model, and the alignment of the model with requirements, is a significant advantage. System requirements, both old and new, can be represented directly using a BP model. This model, together with the revised PN implementation, can be checked for equivalence using the method described in Section 6.4. An equivalence, in such case, indicates an alignment between requirements (old and new) and the revised implementation.

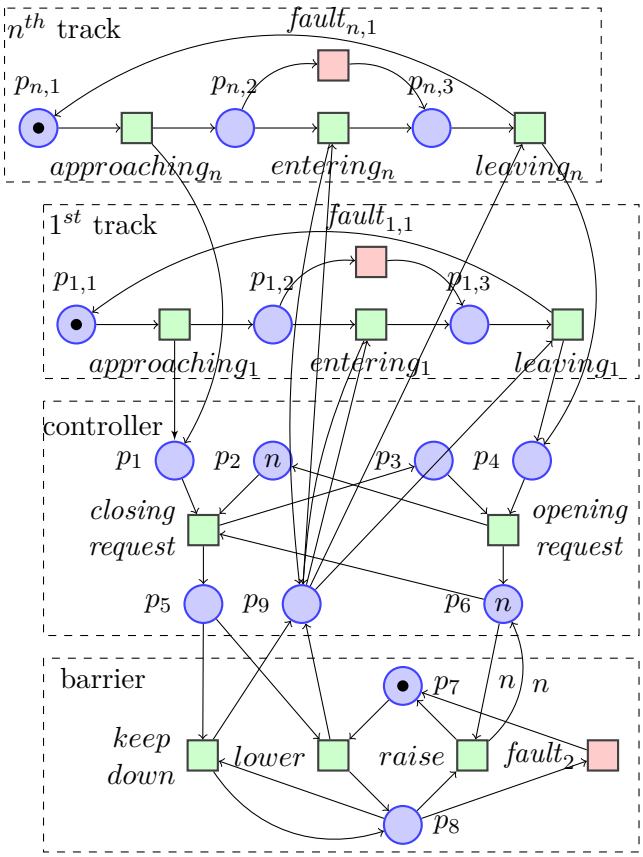


Figure 6.7: Fault extension of the PN multi-track model. The extension consists of additional arcs to satisfy the original requirements [59].

```
@b_thread
def fault_entering(i):
    while True:
        yield {waitFor: Approaching(i)}
        yield {request: Entering(i, fault=True)}

@b_thread
def fault_raise():
    while True:
        yield {waitFor: Lower()}
        yield {request: Raise(fault=True)}
```

Listing 6.3: The added faults b-threads. The b-threads were added incrementally, maintaining the original semantics of the original system.

6.5 Translational Semantics from PN to BP

The notion of equivalence, as presented in Section 6.4, can be useful in bridging the gap between system specification and its PN implementation with the help of BP. System requirements can be represented directly using BP, while the implementation in PN. Hence an equivalence between the two models, in such case, can indicate an alignment between requirements and implementation. This approach "eases" the transition for DES modelers, and maintains the current advantages of PN. While the automata, required for comparison, can be computed directly from PN, we now show a more viable alternative, that is more useful in practice - a direct translation of PN to BP. With this, tools can allow for a uniform modeling environment where all modeling artifacts are translated to a common language.

Taking advantage of BP's modularity and flexibility, we can implement the dynamics of each place in the PN in a separate b-thread. To demonstrate, an example of p_2 (depicted in Figure 6.3) suggested translation, is presented in Listing 6.4. The b-thread holds the variable *tokens*, representing the current number of tokens p_2 holds. If it holds no tokens, it forbids the event "Closing Request" from taking place while waiting for an "Opening Request" event, which increases its tokens. If it holds some tokens, it waits for both events and increases or decreases its tokens accordingly. The suggested translation is general and can be applied to all places in the PN. Combining all places b-threads in a b-program, in addition to a simple auxiliary b-thread requesting all possible events each round (demonstrated in Listing 6.5), yields a behavior consistent with the entire PN dynamics.

```
@b_thread
def p_2():
    tokens = 1
    while True:
        if tokens < 1:
            yield { waitFor: OpeningRequest(), block:ClosingRequest()}
            tokens += 1
        else:
            e = yield {waitFor: [ClosingRequest(), OpeningRequest()]}
            if e == ClosingRequest():
                tokens -= 1
            else:
                tokens += 1
```

Listing 6.4: p_2 translated b-thread. Each place in a PN model can be implemented in a separate b-thread.

Although a translation from BP to PN is not necessary in the context of this chapter, it should be noted that such can be made directly. An automaton representing the behavior of the model, such as the one depicted in Figure 6.5, can be generated. This automaton can of course be viewed as a special case of a simple PN of a single token passing from states (or places).

```
@b_thread
def auxiliary():
    while True:
        yield {request: [ClosingRequest(), OpeningRequest()]}
```

Listing 6.5: Auxiliary translated b-thread

6.6 Discussion and Future Work

In general, there is a qualitative difference between BP and PN: BP focuses on breaking systems to requirements, and PN is focused on specifying the components of a system and how they interact. While both approaches have many merits, we argue in this chapter that the former is better for describing discrete event systems where the details of the implementation are usually of less importance to stakeholders. In this chapter we proposed to model the system in two phases: (1) Model the requirements with BP, allowing for a direct specification, execution, and verification of the requirements; and (2) Implement the requirements with PN.

We showed that the suggested additional phase can assist DES modelling. It separates the requirements from the implementation, thus helping modelers to avoid premature implementation decisions. Additionally, it supports a modular specification approach where each module isolates a specific aspect of the system behavior. The suggested phase allows for verifying different properties of the requirements in a compositional way, thus handling large search spaces. Our proposed method supports the current PN-based practices for DES and thus maintaining its advantages. Further, we give transnational semantics from the BP model to the PN, and vice versa.

We believe that the BP and PN modelling frameworks complement each other. Thus a combination of the two can help and improve DES modelling. Future research directions include integrating concepts from the two for system modelling. As a first step, an automatic translation of a PN to BP and vice versa seems like a promising direction.

Chapter 7

Conclusion

In this thesis, we extended the Behavioral Programming paradigm and explored its potential in software system engineering. We showed how these extensions can simplify and improve software development processes in various domains. We compared BP with our extensions to the baseline BP and other development techniques. Many examples given in this thesis showed that software developed with the proposed methods is better aligned with the specification of systems, as the programmer perceive them, and thus is easier to develop and maintain.

We started by presenting BPpy, a stable and unified implementation of BP in Python. This implementation was used as an infrastructure for the research presented here and is already being used by others in our research group and outside of it. While it is not the essence of this thesis, we believe its contribution is noteworthy. Further, it corresponds with one of the thesis's main goals, making BP more accessible and usable for different applications.

We continued detailing an experimental protocol for a more natural and abstract system modelling through a BP based implementation of a controller for a player in a simplified RoboCup-type game. The implementation is a part of a paper published in MDETOOLS'19 workshop on model-driven engineering tools [6]. In the protocol, a combination of BP and Deep Reinforcement Learning allows for giving abstract instructions to a system and for teaching it to follow them. We showed that this combination can facilitate a simplification of the model. Nonetheless, further research is required to better formalize it and demonstrate its contribution.

We later demonstrated a method allowing the direct specification of liveness requirements in BP easily and intuitively. By integrating Reinforcement Learning to the event selection mechanism, we showed that the specified liveness requirements can be enforced without adding unnecessary external constraints. We demonstrated how this extension can enhance the programming experience and how it is simple and intuitive for software developers. We chose to focus on cases where liveness properties relate to the state of each b-thread separately. Future work will aim at generalizing the proposed method to fit the case where liveness properties relate to the state of the entire b-program. Additionally, further research is needed to establish a more reliable learning algorithm

for our method that offers the same level of generality.

Lastly, we proposed a methodology for Discrete Event Systems (DES) modelling using BP and Petri Nets. We argued that the Petri Net formalism for modeling DES is driving modelers to premature design decisions that lead to incomplete and inaccurate models that are hard to generalize and maintain. The methodology offers to model system requirements with BP prior to implementing it with Petri Net, to avoid premature implementation decisions. Future research directions include integrating concepts from BP and Petri Nets for system modelling. As a first step, an automatic translation of a Petri Nets to BP and vice versa seems like a promising direction.

The quest for improving programming languages and methodologies is longstanding and is still far from over. We hope that the work presented here is a step in the journey ahead.

Bibliography

[1] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Communications of the ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[2] J. C. Mogul, "Emergent (mis) behavior vs. complex software systems," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 293–304, 2006.

[3] S. Dalal, M. Lyu, and C. Mallows, "Software reliability," *Wiley StatsRef: Statistics Reference Online*, 2014.

[4] N. R. Jennings, "An agent-based approach for building complex software systems," *Communications of the ACM*, vol. 44, no. 4, pp. 35–41, 2001.

[5] T. Yaacov, "Bppy: Behavioral programming in python." `https://github.com/bThink-BGU/BPpy`, 2020.

[6] A. Elyasaf, A. Sadon, G. Weiss, and T. Yaacov, "Using behavioural programming with solver, context, and deep reinforcement learning for playing a simplified robocup-type game," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 243–251, IEEE, 2019.

[7] D. Harel, A. Marron, and G. Weiss, "Programming coordinated behavior in java," in *European Conference on Object-Oriented Programming*, pp. 250–274, Springer, 2010.

[8] W. Damm and D. Harel, "Lscs: Breathing life into message sequence charts," *Formal methods in system design*, vol. 19, no. 1, pp. 45–80, 2001.

[9] D. Harel and R. Marelly, *Come, let's play: scenario-based programming using LSCs and the play-engine*, vol. 1. Springer Science & Business Media, 2003.

[10] M. Bar-Sinai, G. Weiss, and R. Shmuel, "Bpjs: an extensible, open infrastructure for behavioral programming research," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pp. 59–60, 2018.

[11] B. Shimony, I. Nikolaidis, P. Gburzynski, and E. Stroulia, "On coordination tools in the picos tuples system," in *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, pp. 19–24, 2011.

[12] M. Bar-Sinai, A. Elyasaf, A. Sadon, and G. Weiss, "A scenario based on-board software and testing environment for satellites," in *The 59th Israel Annual Conference on Aerospace Sciences (IACAS), 2019*, 2019.

[13] D. Harel, R. Lampert, A. Marron, and G. Weiss, "Model-checking behavioral programs," in *Proceedings of the ninth ACM international conference on Embedded software*, pp. 279–288, 2011.

[14] D. Harel, G. Katz, A. Marron, A. Sadon, and G. Weiss, "Executing scenario-based specification with dynamic generation of rich events," in *International Conference on Model-Driven Engineering and Software Development*, pp. 246–274, Springer, 2019.

[15] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, pp. 305–343, Springer, 2018.

[16] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[17] R. S. Sutton, A. G. Barto, *et al.*, *Introduction to reinforcement learning*, vol. 135. MIT press Cambridge, 1998.

[18] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[21] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun, "Pedestrian detection with unsupervised multi-stage feature learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3626–3633, 2013.

[22] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2011.

[23] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, Ieee, 2013.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[25] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[26] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[27] N. Eitan and D. Harel, "Adaptive behavioral programming," in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pp. 685–692, IEEE, 2011.

[28] O. M. Weinstock, "Online search in behavioral programming models," *Proceedings of the ACM Student Research Competition at MODELS*, pp. 58–63, 2015.

[29] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.

[30] B. Alpern and F. B. Schneider, "Defining liveness," *Information processing letters*, vol. 21, no. 4, pp. 181–185, 1985.

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

[32] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[33] A. Giua and M. Silva, "Modeling, analysis and control of discrete event systems: a petri net perspective," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 1772–1783, 2017.

[34] P. Sobociński, "Representations of petri net interactions," in *International Conference on Concurrency Theory*, pp. 554–568, Springer, 2010.

[35] P. Sobocinski, U. Montanari, H. Melgratti, and R. Bruni, "Connector algebras for c/e and p/t nets' interactions," *Logical Methods in Computer Science*, vol. 9, 2013.

[36] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel, "Compositional modeling of reactive systems using open nets," in *International Conference on Concurrency Theory*, pp. 502–518, Springer, 2001.

[37] R. Eshuis and J. Dehnert, "Reactive petri nets for workflow modeling," in *International Conference on Application and Theory of Petri Nets*, pp. 296–315, Springer, 2003.

[38] D. Harel, A. Marron, G. Wiener, and G. Weiss, "Behavioral programming, decentralized control, and multiple time scales," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pp. 171–182, 2011.

[39] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*, pp. 103–120, Springer, 2012.

[40] N. Cardozo, S. González, K. Mens, R. Van Der Straeten, J. Vallejos, and T. D'Hondt, "Semantics for consistent activation in context-oriented systems," *Information and Software Technology*, vol. 58, pp. 71–94, 2015.

[41] E. Best and M. Koutny, "Petri net semantics of priority systems," *Theoretical Computer Science*, vol. 96, no. 1, pp. 175–215, 1992.

[42] K. Jensen and L. M. Kristensen, *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.

[43] G. Katz, "Towards repairing scenario-based models with rich events," *arXiv preprint arXiv:2101.03504*, 2021.

[44] M. Lutz, *Programming python*. " O'Reilly Media, Inc.", 2001.

[45] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[46] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[47] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[48] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss, "On composing and proving the correctness of reactive behavior," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–10, IEEE, 2013.

[49] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

[50] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[51] M.-P. B. Schrader, "gym-sokoban." `https://github.com/mpSchrader/gym-sokoban`, 2018.

[52] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 219–251, 2001.

[53] S. Racanière, T. Weber, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, *et al.*, "Imagination-augmented agents for deep reinforcement learning.," in *NIPS*, pp. 5690–5701, 2017.

[54] K. Katayama, T. Koshiishi, and H. Narihisa, "Reinforcement learning agents with primary knowledge designed by analytic hierarchy process," in *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 14–21, 2005.

[55] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[56] J. Magee and J. Kramer, "Model-based design of concurrent programs," in *Communicating Sequential Processes. The First 25 Years*, pp. 211–219, Springer, 2005.

[57] N. G. Leveson and J. L. Stolzy, "Safety analysis using petri nets," *IEEE Transactions on software engineering*, no. 3, pp. 386–397, 1987.

[58] B. Liu, M. Ghazel, and A. Toguyéni, "Of-penda: A software tool for fault diagnosis of discrete event systems modeled by labeled petri nets.," in *ADECS@ Petri Nets*, pp. 20–35, 2014.

[59] M. Ghazel and B. Liu, "A customizable railway benchmark to deal with fault diagnosis issues in des," in *2016 13th International Workshop on Discrete Event Systems (WODES)*, pp. 177–182, IEEE, 2016.

[60] A. Mazzeo, N. Mazzocca, S. Russo, and V. Vittorini, "A systematic approach to the petri net based specification of concurrent systems," in *Safety-Critical Real-Time Systems*, pp. 3–20, Springer, 1997.

[61] A. Boussif, *Contributions to model-based diagnosis of discrete-event systems*. PhD thesis, Université de Lille1-Sciences et Technologies, 2016.

[62] A. Boussif, B. Liu, and M. Ghazel, "An experimental comparison of three diagnosis techniques for discrete event systems," in *DX'17-28th International Workshop on Principles of Diagnosis*, p. p8, 2017.

[63] G. Liu, C. Jiang, and M. Zhou, "Time-soundness of time petri nets modelling time-critical systems," *ACM Transactions on Cyber-Physical Systems*, vol. 2, no. 2, pp. 1–27, 2018.

[64] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, K.-N. Wu, and W.-C. Chan, "Goal: A graphical tool for manipulating büchi automata and temporal formulae," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 466–471, Springer, 2007.

תקציר

עבודת גמר זו מציגה מספר הרחבות לפרדיגמת התכנות "תכנות התנגותי", במטרה להפוך אותה ליותר נגישה ויעילה לשימושים ליישומים שונים. התרומות של עבודת הגמר הן כדלקמן:

1. תכנות התנהגותי / פייתון: מימוש יציב ואחיד לתכנות התנהגותי בפייתון. מימוש זה שימש כתשתית למחקר שהתבצע ומוצג כאן וכבר משמש חוקרים נוספים בקבוצת המחקר שלנו ומחוצה לה.

2. תכנות התנהגותי / למידת חיזוק עמוקה: פרוטוקול ניסיוני למידול מערכת באופן יותר טבעי ואבסטרקטי. בפרוטוקול, שילוב של תכנות התנהגותי ולמידת חיזוק עמוקה מאפשר מתן הוראות אבסטרקטיות למערכת ולמידה שלה לעקוב אחריהם.

3. תכנות התנהגותי / חיות: שיטה המאפשרת הגדרה ישירה של דרישות חיות בתכנות התנהגותית. על ידי אינטגרציה של למידת חיזוק למנגנון החלטת האירועים, אנו מראים שאפשר לאכוף דרישות חיות בריצת התוכנה מבלי להוסיף אילוצים חיצוניים מיותרים.

4. תכנות התנהגותי / מערכות אירועים דיסקרטים: מתודולוגיה למידול מערכות אירועים דיסקרטים באמצעות תכנות התנהגותי ורשתות פטרי. המתדולוגיה מציעה למדל דרישות מערכת באמצעות תכנות התנהגותי כצעד קודם למימוש ברשתות פטרי על מנת להימנע מהחלטות מימוש מוקדמות מדי.

אנו מראים כיצד שילוב של תרומות אלה משפר את תהליך פיתוח התוכנה על ידי תמיכה בפיתוח טבעי ומצטבר שלה. אנו משווים תכנות התנהגותי עם השיפורים שלנו לתכנות ההתנהגותי הבסיסי ולשיטות פיתוח אחרות. דוגמאות רבות המוצגות בעבודת הגמר הזו מראות שפיתוח תוכנה עם השיטות המוצגות תואם בצורה מיטיבה עם המפרט של המערכת, האופן שבו המפתח תופס אותה ולכן קלה יותר לפיתוח ותחזוק.

המסע לשיפור שפות ומתודולוגיות פיתוח קיים זמן מה ורחוק מלהסתיים. מכיוון שאנו מאמינים שהמשכו של המסע נמצא בשילוב בין שיטות, אנו מתמקדים בשילוב של תכנות התנהגותי עם שיטות וכלים כפי שמתואר ברשימה לעיל.

מילות מפתח: תכנות התנהגותי, פייתון, פתרני אילוצים, למידת חיזוק, חיות, מערכות אירועים דיסקרטים, רשתות פטרי.

אוניברסיטת בן־גוריון בנגב
הפקולטה למדעי הטבע
המחלקה למדעי המחשב

הרחבת תכנות התנהגותי לשיפור הנדסת תוכנה

חיבור לשם קבלת התואר "מגיסטר" בפקולטה למדעי הטבע

מאת תם יעקב

בהנחיית פרופ׳ גרא וייס וד״ר אחיה אליסף

י׳ תמוז ה׳תשפ״א