Using behavioural Programming with Solver, Context, and Deep Reinforcement Learning for Playing a Simplified RoboCup-type Game

Achiya Elyasaf, Aviran Sadon, Gera Weiss, Tom Yaacov
Ben-Gurion University of the Negev, Beer-Sheva, 8410501, Israel
Email: {achiya, geraw}@bgu.ac.il , {sadonav, tomya}@post.bgu.ac.il

***Abstract*—We describe four scenario-based implementations of controllers for a player in a simplified RoboCup-type game. All four implementations are based on the behavioural programming (BP) approach. We first describe a simple controller for the player using the state-of-the-art BPjs tool and then show how it can be extended in various ways. The first extension is based on a version of BP where the Z3 SMT solver is used to provide mechanisms for richer composition of modules within the BP model. This allows for modules with higher cohesion and lower coupling. It also allows incrementality: we could use the scenarios we developed for the challenge of MDETOOLS'18 and extend the model to handle the new system. The second extension of BP demonstrated in this paper is a set of idioms for subjecting model components to context. One of the differences between this year's challenge and the challenge we dealt with last year is that following the ball is not the only task that a player needs to handle, there is much more to care for. We demonstrate how we used the idioms for handling context to parametrize scenarios like "go to a target" in a dynamic and natural fashion such that modelers can efficiently specify reusable components similar to the way modern user manuals for advanced products are written. Lastly, in an attempt to make the instructions to the robot even more natural, we demonstrate a third extension based on deep reinforcement learning. Towards substantiating the observation that it is easier to explain things to an intelligent agent than to dumb compiler, we demonstrate how the combination of 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.**

I. Introduction

As robots are expected to be more sophisticated and autonomous, robot developers require tools that allow them to effectively develop and analyze robust robot control software. In this paper we demonstrate the use of recent modelling techniques and tools that we have developed for this purpose. The demonstration is done by describing parts of a small model that we have developed to control a robot in a virtual game called RoboSoccer that was proposed as a challenge for the participants of the MDETOOLS'19 conference (see http://mdetools.github.io/mdetools19/challengeproblem.html). 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.

All the models described in this paper are built using variants of the scenario-based modelling approach called *behavioural Programming* (BP) [1]. As demonstrated in the examples below, and other case studies and theoretical work [2]–[4], BP allows for modular specifications where each module isolates a specific aspect of the robot's behaviour. This is made possible by protocols that allow the components of a model to express the specific aspect of the behaviour that they represent. With these protocols, each component can dynamically specify what it *wants* the robot to do and what it *wants to block* the robot from doing. An application-agnostic execution mechanism repeatedly collects these specifications, chooses actions that are consistent with all the specifications, executes them, and continuously informs the components of each selection. See Section II for further details.

Previous demonstrations of BP include a show case of a fully functional nano-satellite [5], a development tool that allows an integration of BP models with imperative code [6], a model-checking tool for verifying the correctness of BP models [7], and implementations in a variety of languages, including Live Sequence Charts (LSC) [8], [9], JavaScript [6], Java [10], C [11] and more. Research results on BP cover, among others, run-time lookahead (smart playout) [12], methods for compositional verification [13], for synthesis [14] and for interactive analysis of unrealizable specification [15].

In this paper we demonstrate three extensions that we are currently developing on top of the base BP protocol proposed in [10]. The extensions are briefly described in the next paragraphs and are further elaborated in the following sections. The complete code for all of the controllers presented in this paper can be found in https://github.com/bThink-BGU/Papers-2019-MDETools.

A base BP model for controlling the robot in RoboSoccer is presented in Section IV. This version of the controller uses the baseline BP protocol proposed in [10] and described in Section II. It demonstrates how the instructions to the robot can be specified using simple modules, each representing a separate behaviour aspect of the robot.

The first extension is presented in Section V. 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 behaviour 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.

A second extension is presented in Section VI. There, we demonstrate how a recently proposed protocol for managing context [16] can serve for subjecting behaviours to contexts and for maintaining information that serves the components of the model. In the RoboSoccer game this allows to reuse behaviours: we can, for example, model "go to ball" and "go to goal" as two instances of the same behaviour only with different parameters. It also allows to ease the management and the identification of the code that corresponds to the requirements for the behaviour in a certain context, e.g., the set of requirements that specify the behaviour when the ball is free or the set that specify how to behave when we are defending our goal. We demonstrate the usefulness of having a formal definition of: (1) different contexts, (2) the information held in each context, and (3) the relationships between the components of the code and context.

A third extension is presented in Section VII. There, we give an example of how the instructions that a modeller gives to a robot 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 interprets our commands intelligently than it is to give instructions to a dumb compiler. Specifically, we show that a combination of BP and deep reinforcement learning allows for specifying behaviour 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 paper continues as follows. In Section II, we present the core concepts of BP and explain, using simple examples, how the baseline BP protocols works. In Section III we provide, for completeness, a short description of the RoboSoccer game. In Section IV, we describe parts of a controller that uses only the baseline BP protocol. This controller serves as the basis for the rest of the controllers in this paper. The controllers that use the extensions to the baseline protocol are then described in Section V, Section VI, and Section VII as said above.

II. behavioural Programming

The behavioural Programming (BP) [1] paradigm focuses on reactive behaviours: how a system reacts to its environment, and how it changes its internal states when certain events happen. When creating a system using BP, developers specify a set of scenarios that may, must, or must not happen. 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 behavioural program (*b-program*). During run-time, all b-threads participating in a b-program are combined, yielding a complex behaviour that is consistent with all said b-threads. Unlike other paradigms, BP does not force the developers to pick a single behaviour for the system to use. Rather, the system is allowed to choose any compliant behaviour. This allows the run-time to optimize program execution at any given moment, e.g., based on available resources. The fact that all possible system behaviours comply with the b-threads (and thus with the system requirements), ensures that whichever behaviour is chosen, the system as a whole will perform as specified.

In [10], 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 submits 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 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. The rest 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. This cycle is presented in the *BP Cycle* frame of Figure 1.

To make these concepts more concrete, we now turn to a tutorial example of a simple b-program, written using BPjs — an environment for running behavioural programs written in JavaScript [6]. The example presented in this section is an adaptation of one of the first sample programs presented in [10] (the HOT/COLD example).

Consider a system with the following requirements:

1) When the system loads, do 'A' three times.
2) When the system loads, do 'B' three times.

Listing 1 shows a b-program (a set of b-threads) that fulfills these requirements. It consists of two b-threads,

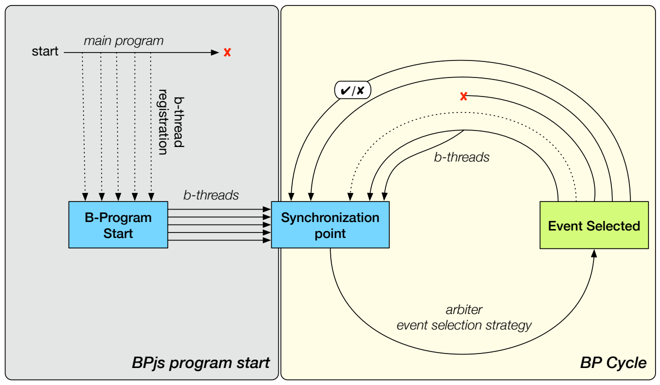

Fig. 1. Life cycle of a b-program using BPjs. The execution starts with a regular JavaScript program, which registers b-threads. Once that program terminates, all b-threads are started concurrently. B-threads repeatedly execute an internal logic and then synchronize with each other, by submitting a synchronization statement to a central event arbiter. Once all b-threads have submitted their statements, the central event arbiter selects an event that was requested and was not blocked. B-threads that either requested or waited for this event are resumed, while the rest of the b-threads remain paused for the next cycle. During their execution phase, b-threads can terminate, register new b-threads, and perform assertions.

added at the program start-up. One b-thread, namely `Do-A`, is responsible for fulfilling requirement #1, and the second b-thread, namely `Do-B`, fulfills requirement #2.

```
bp.registerBThread("Do-A", function(){
  bp.sync({request: bp.event("A")});
  bp.sync({request: bp.event("A")});
  bp.sync({request: bp.event("A")});
});
bp.registerBThread("Do-B", function(){
  bp.sync({request: bp.event("B")});
  bp.sync({request: bp.event("B")});
  bp.sync({request: bp.event("B")});
});
```

Listing 1. A b-program that do ‘A’ and ‘B’ three times each. The order between ‘A’ and ‘B’ events is arbitrary.

The program's structure is aligned with the system requirements. It has a single b-thread for each requirement, and it does not dictate the order in which actions are performed (e.g., the following runs are possible: AABBAB, or ABABAB, etc.). This is in contrast to, say, a single-threaded JavaScript program that would have to dictate exactly when each action should be performed. Thus, traditional programming paradigms are prone to over specification, while behavioural programming avoids it.

While a specific order of actions was not required originally, in some cases, this behaviour may represent a problem. Consider for example an additional requirement that the user detected after running the initial version of the system:

3) Two actions of the same type cannot be executed consecutively.

While we may add a condition before requesting ‘A’ and ‘B’, the BP paradigm encourages us to add a new b-thread for each new requirement. Thus we add a b-thread, called `Interleave`, presented in Listing 2.

```
bp.registerBThread("Interleave", function() {
  while(true) {
    bp.sync({waitFor: bp.event("B"),
             block: bp.event("A")});

    bp.sync({waitFor: bp.event("A"),
             block: bp.event("B")});
  }
});
```

Listing 2. A b-thread that ensures that two actions of the same type cannot be executed consecutively, by blocking and additional request of ‘A’ until the ‘B’ is performed, and vice-versa.

The `Interleave` b-thread ensures that there are no repetitions. It does so by forcing an interleaved execution of the performed actions — first ‘A’ is blocked until ‘B’ is executed, then ‘B’ is blocked until ‘A’ is executed. This is done by using the `waitFor` and `block` idioms. Note that this b-thread can be added and removed without affecting other b-threads. This is an example of a *purely additive* change, where the system behaviour is altered to match a new requirement without affecting the existing behaviours. While not all changes to a b-program are purely additive, many useful changes are, as demonstrated in the examples below.

III. The Simplified RoboCup-Type Game

This paper is part of the MDETOOLS'19 workshop on model-driven engineering tools. 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 is adopted from the text that appears on the web site of the workshop at https://mdetools.github.io/mdetools19/challengeproblem.html.

The organizers of the workshop created a simulation of a simplified RoboCup-type game inspired by https://ssim.robocup.org as shown in Figure 2. The simulation 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.

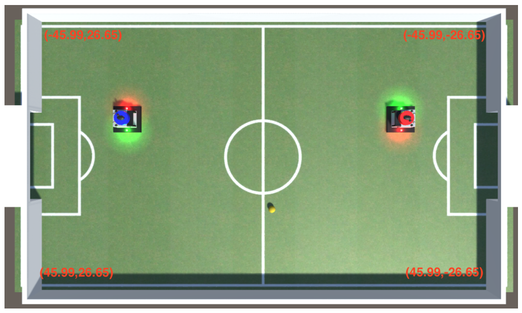

Fig. 2. The RoboSoccer game.

At the beginning of the simulation, the ball lies on a kick off position in the field 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.

IV. A Baseline Controller in BPjs

In this section, we present a simple controller that drives the robot to the ball. We will later improve this model using tools that we have developed to enhance BP. This section is given as a baseline for the developments that we will discuss in the following sections.

As we will elaborate in Section V below, even though the RoboSoccer robots are "mechanically" different from the rovers that we were challenged to control last year, thanks to the incrementality feature of BP, we were able to use the rover b-threads also for the RoboSoccer game. Essentially, following the ball is very similar to following the leader rover.

A model for driving the player (i.e., red robot) to the ball is given in Listing 3. This is a very simple BP model consists of three b-threads called "MoveTowards-Ball", "SpinToBall", and "BallSuction". The model is an adaptation of a model we have presented for the rover challenge that appeared in MDETOOLS'18 (https://mdetools.github.io/mdetools18/challengeproblem.html).

All the three b-threads start by waiting for a state of the game that matches a condition. The first two wait for a state that meets the `BallFree` condition and the third waits for a state that meets the `BallNearPlayer` condition. This is because we only show here the b-threads that handle the logic of approaching and catching the ball. When a game `StateUpdate` event that matches the relevant condition is triggered (by a mechanism that is not shown here), its value is returned from the `bp.sync(...)` call and stored in the `state` variable. This code is common to all three b-threads. Each b-thread then responds to the new state, as explained in the paragraphs below, and returns to wait for another state event that matches its condition in an infinite loop.

The "MoveTowardsBall" b-thread takes care of controlling the speed of approaching the ball. It first computes the distance between the robot and the ball using the function `distanceFromPlayerToBall`. Then, it decides if the distance is such that a gradient approach is needed using the `needGradient` function and then it requests a `moveForward` event with full power or with a gradient computed by the `gradient` function. The code for the functions used in this b-thread is removed for brevity. Note that this type of abstraction is allowed because we use a full fledged scripting language in our modeling tool.

```
bp.registerBThread("MoveTowardsBall", function() {
  while(true) {
    var state = bp.sync({waitFor: StateUpdate.BallFree});
    var distance = distanceFromPlayerToBall(state);
    if(needGradient(distance))
      bp.sync({request: moveForward(gradient(distance))});
    else
      bp.sync({request: moveForward(MAX_PWR)});
  }
});

bp.registerBThread("SpinToBall", function() {
  while(true) {
    var state = bp.sync({waitFor: StateUpdate.BallFree});
    var degree = degreeFromPlayerToBall(state);
    if (degree < -MIN_DEGREE)
      bp.sync({request: spin(MAX_SPIN)});
    else if (degree > MIN_DEGREE)
      bp.sync({request: spin(-MAX_SPIN)});
    else
      bp.sync({request: spin(0)});
  }
});

bp.registerBThread("BallSuction", function() {
  while(true) {
    bp.sync({waitFor: StateUpdate.BallNearPlayer}));
    bp.sync({request: suction()});
  }
});
```

Listing 3. A base controller in BPjs

The "SpinToBall" b-thread is responsible for controlling the direction that the robot moves to. It checks if the angle of the vector from the robot to the ball is above some threshold and then requests to spin with either `MAX_SPIN` or its negation depending on whether the angle is positive or negative. If the angle is small enough, the b-thread requests to stop the spin.

The "BallSuction" b-thread simply requests to activate ball suction whenever the player is near the ball and the ball is free.

This model shows how a simple controller for the game is implemented using the core features of the BPjs library. In the next sections, we will discuss other implementations of controllers for the same tasks using new modelling features that we have recently developed.

V. Using a Solver for Real Valued Variables

In this section, we describe an extension of the controller for the red robot, using a recent version of BP that we have developed, called BP-Z3. The main difference between the modelling variant presented here and the one described in the previous section, is that here we apply the Z3 satisfiability modulo theory (SMT) solver [17] for event selection. Another difference, less substantial, is that we are using Python instead of JavaScript as the underlying scripting language. For technical reasons, in this version we use the keyword `yield` for specifying the synchronization points, instead the keyword `bp.sync` used above.

The motivation for using a solver-based event selection protocol is as follows. 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. While BPjs allows for attaching rich data, including numerical values, to the events, its event selection is discrete. Specifically, the event selection protocol in BPjs goes by enumerating all events that were requested (usually in a random order) and selecting the first one that is not blocked. This dictates that we can only allow for enumerable requested event sets. For robots that require rich commands with requirements that pose complex relations between the variables in the events, this may cause significant complications.

```
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. Invariant b-thread for the solver-based controller. Note that we are using Python here and that in this version of BP, `bp.sync` is replaced by `yield`. Also, the specification of what a b-thread waits for, what it requests, and what it blocks are given as logical and arithmetic constraints.

To cope with the issue raised in the preceding paragraph, and other challenges that were raised in other contexts, we have developed an experimental implementation of a BP engine that allows b-threads to specify complex constraints in their specification of what they request, block, or wait for. In each synchronization round, after collecting all constraints, as done also in BPjs, the new execution mechanism invokes the Z3 SMT solver to find an assignment to the data fields embedded in the events (called variables hereafter). The assignment is then returned by the `yield` command of the b-thread. For example, the `forward` variable in Listing 4, 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, because in this version of BP the default is `waitFor: true`, i.e., a b-thread is awaken after each event unless something else is specified.

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 5, 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 6. 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), 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 then `TOO_FAR` and the robot is facing the ball, the execution of these b-threads will yield the event (i.e., the solver assignment) `{forward:MAX_PWR, spin:0, ...}` that will be assigned to the variable `m`. 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 behaviour that the designers and the users perceive. It also allows for more effective modelling of systems that require focus on numerical and Boolean fields in the events.

```
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 5. The `MoveTowardsBall` b-thread using the solver based BP. 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.

```
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 6. Using the solver based semantics to allow the b-thread that controls the spin to only specify the direction of the spin, leaving the specification of the exact value to other b-threads.

The above b-threads can be extended to a functional controller for the red robot, similar in nature to the one that uses the BPjs library presented in the previous section. We actually have been experimenting with two different controllers for this mission. The controller presented here is the one that is more similar to the BPjs based controller, hence the choice to put it in the front. But the other controller is also interesting to report upon because it reassembles the controller we developed for the MDETOOLS'18 where we were challenged to control a rover that follows another rover in a Martian like terrain. The rover was controlled by setting the speeds of the left and of the right wheels separately. Using different commands to these speeds, one could generate turn and forward movements. The reason that we are describing this here is because we were able to use the code that we have developed for the rovers using the BP+Python+Z3 tools (which is not described in the submission to MDETOOLS'18 because it was only done after the workshop). The BP+Python+Z3 based controller for the rover challenge is described in a paper presented at MODELSWARD'19 [18]) and we could use it as-is for the RoboSoccer challenge by adding b-threads that model the relations between the variables solved by the older code to the variables needed for RoboSoccer. This is possible because following the ball is similar to following a rover and because the solver-based event selection protocol allows for specifying complex relations between variables and for flexibility in how engineers can break their models to modules. In the controller not show here, we separated the control logic to two main scenarios. The first is reaching the ball, where we used last year's logic to "follow" the ball. Once the robot reaches the ball the new target we are after, that acts as a "leader", is the goal. We also added b-threads to handle the suction variable. See https://github.com/bThink-BGU/Papers-2019-MDETools.

VI. Using ConBPjs for Context Awareness

While b-thread can be completely independent in some specifications, requirements for many reactive systems are *context* dependant requiring shared information and data. Contexts can be defined as information that can be used to characterize the situation of entities or processes in a system [19]. The term *context awareness* refers to the system's ability to use the context information [19]. In this section we show how a protocol we have designed to allow for context awareness in BP models, is used to improve our controller for the RoboSoccer game. In this version of BP we go back to using JavaScript and `bp.sync(...)`, as in Section IV.

As an example of a context in the game, note that the player should act differently when the target is the ball or the goal. It should also act differently when the ball is possessed by the opponent and when the ball is free. Each b-thread in Listing 3 starts with detecting states that meet the `BallFree` of `BallNearPlayer`. The event returned from `bp.sync` is assumed to contain the entire context of the system, even though none of the b-threads uses the entire state. In fact, each b-thread in the BP paradigm is responsible of only one aspect of the system's behaviour, thus giving it an access to the entire state undermines the basic idea in a way. The solution we have implemented in Listing 3 of hiding the details in the `distanceFromPlayerToBall(state)` and `degreeFromPlayerToBall(state)`, is a viable ad-hoc solution , but we have recently developed a more systematic and formal approach to the state awareness, as demonstrated in this section. Context awareness is not just another name for the environment, it is a method for creating an abstraction that clearly and intuitively defines aspects of the state of the environment and of the internal states of the system, that are most important for subjecting the behaviour upon.

In order to allow context awareness in BP, we developed an extension to the behavioural programming paradigm with idioms for explicitly defining contexts and referencing them. This approach was first presented in [16], where we extended the Live Sequence Charts (LSC) language with context idioms, and presented a context-oriented behavioural programming methodology for developing context-aware systems. In this paper we will demonstrate an extension that allows for context awareness in BPjs.

In Listing 7, each b-thread is bound to a certain context. When moving towards a target, whether the target is the ball or the goal, we only need to know the distance and the angle from the player to target. Thus the b-threads are bound to the `MoveTowardsTarget` context that keeps the relevant data only. The `CTX.subscribe` method ensures that the bound code is executed whenever a new instance of the context is created.

We handle the contextual data in terms of a relational data model using a real-time database for maintaining the contextual data. The context 'select' queries are automatically translated to database views that allow for triggering a stored procedure whenever a record is added or removed from a view. Context 'update' commands are translated to stored procedures that, when invoked, update the database as required (by adding, deleting, and changing objects and object relations). We handle these translations in a data access layer, while the behavioural specification (i.e., the JavaScript code), is handled in a business logic layer.

The code in Listing 7 only queries the database using 'select' queries. For example, the `MoveTowardsTarget` query is translated to `SELECT distanceFromPlayer↩ , degreeFromPlayer FROM target`. The use of 'update' commands is demonstrated in Listing 8. The `UpdateBallPossessing` command, is translated to `UPDATE referee SET possessing=:playerName`, where `playerName` is a parameter. This update triggers several view changes (i.e., new context instances and the termination of others), for example, if the player had the ball and it has now become free, we will have changes in the `PlayerPossessTheBall` context and in the `BallIsFree` context.

Finally, we did not, until now, take the opponent into our considerations. We can use the context idioms for

```
CTX.subscribe("MoveTowardsTarget", "MoveTowardsTarget", function(target) {
  var distance = target.distanceFromPlayer;
  if(needGradient(distance))
    bp.sync({request: moveForward(gradient(distance))});
  else
    bp.sync({request: moveForward(MAX_PWR)});
});

CTX.subscribe("SpinToTarget", "MoveTowardsTarget", function(target) {
    var degree = target.degreeFromPlayer;
  if (degree < -MIN_DEGREE)
    bp.sync({request: spin(MAX_SPIN)});
  else if (degree > MIN_DEGREE)
    bp.sync({request: spin(-MAX_SPIN)});
  else
    bp.sync({request: spin(0)});
});

CTX.subscribe("BallSuction", "BallIsFreeNearPlayer", function(c) {
    bp.sync({request: suction()});
});
```

Listing 7. An improvement using ConBPjs — an extension to BPjs with idioms for explicitly defining contexts and referencing them.

```
CTX.subscribe("UpdatePossession", "Playing", function(referee) {
  var playerName = bp.sync({ waitFor: possession() }).name;
  bp.sync({ request: CTX.UpdateEvent("UpdatePossession", { "possession": playerName }) });
});

CTX.subscribe("UpdateTarget", "BallIsFree", function (referee) {
  var ctxEndedEvent = CTX.ContextEndedEvent("BallIsFree", referee);
  while(true) {
    var ball = bp.sync({waitFor: StateUpdate.ANY, interrupt: ctxEndedEvent}).ball;
    bp.sync({
      request: CTX.InsertEvent(new Target("ball", distanceFromPlayer(ball), degreeFromPlayer(ball)))
    });
  }
});
```

Listing 8. Updating the context

applying different strategies depending on the opponent state and behaviour. The context idioms increase the modularity and incrementally of BP. For example, if the ball is free and the opponent is nearer to it than our robot, we could defend our gate instead of moving towards the ball (assuming, of course, that the game engine provides us the information about the opponent, which is not available at the current version of the engine).

VII. Using Deep Reinforcement Learning for Simplicity and Robustness

In this section we present an attempt to make the instructions described in Section V even more natural and abstract. A motivating example can be found in the `move_towards_ball` b-thread. As described in Listing 5, 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 a reinforcement learning (RL) mechanism for this task. This setting allows the instructions to be simpler, more robust, and easier to maintain when the simulation changes. The idea of merging BP with RL was first presented by Eitan at. al. [20] in 2011. Here, we show that with a combination of a rich solver and deep 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 »Ї direct interaction with its environment, without requiring exemplary supervision or complete models of the environment [21]. In the standard RL model, an agent is connected to its environment via perception and action, as depicted in Figure 3. On each step t of an interaction, the agent receives as input some indication of the current state, s_t, of the environment. The agent then chooses an action, a_t, to generate as output. The action changes the state of the environment, and the value of this state transition is communicated to the agent through a reward signal, r_t. The goal is to find a *policy*, which is a function that gives the best action an agent can take, given the state of the environment, such that the long term reward is maximized.

In our setting, depicted in Figure 4, the agent interacts with the b-program, that encapsulates the environment, in

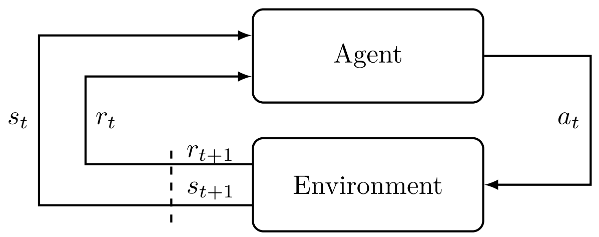

Fig. 3. A standard RL model

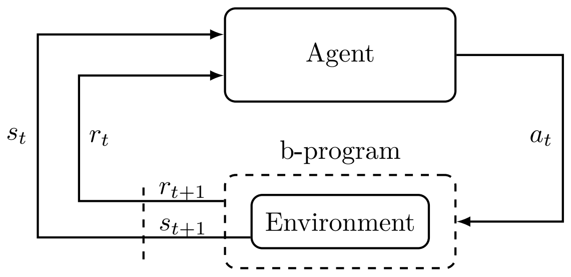

Fig. 4. A modified behavioural RL model

our case — the RoboSoccer 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 in order to modify the robot's behaviour. In this challenge, we targeted the task of grabbing the ball. As shown in Listing 9, 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.

```
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 9. The modified `get_ball_reward` b-thread. The agent's reward in each step is defined by `is_ball_in_robot`.

In order to assist with the task of ball grabbing, the b-threads that are related for 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 10, 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`. The `spin_to_ball` b-thread, which controls the direction of the spin, was modified in the same manner, as shown in Listing 11. 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.

```
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 10. 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`

```
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 11. The modified `spin_to_ball` b-thread. the direction of the spin is controlled by the RL agent action `need_to_spin`.

Learning to control agents directly from high-dimensional sensory inputs, like the simplified RoboCup-type simulation state, is one of the long-standing challenges of reinforcement learning. Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in various domains [22]. In order for these techniques to be beneficial for RL with sensory data, reinforcement-learning approaches are augmented with deep neural networks (DRL). One of the most successful DRL algorithms is *Deep Q Network* (DQN) [22]. In its raw form, DQN uses a multilayer perceptron network for the policy function approximation.

In our implementation, which uses the DQN implementation of [23], 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 in order to modify the game controler. Note that we are not just using DRL to achieve 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 (`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.

VIII. Conclusion, Discussion, and Future Work

We have demonstrated four BP approaches to program the red robot in the RoboSoccer game:

1) We started with a model that uses the base BP protocol to coordinate b-threads representing aspects of the actions needed to hold the ball and score a goal.
2) We then demonstrated how similar behaviour 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.
3) We also demonstrated how context-aware modelling is used to improve the model. Specifically, we showed how a disciplined methodology, supported by tools, for sharing information between the b-threads and for coordinating mode transitions, is useful for simplifying and for generalizing the model.
4) We concluded by showing how deep reinforcement learning protocols can facilitate further simplification of the model, by allowing programmers to specify abstract behaviour and have an automatic tool sort the implementation details.

The goal of this paper was to present how the RoboSoccer challenge can be tackled using BP, towards comparison with other tools that will be presented in the conference. As a future work, after the conference, we hope to be able to compare our approaches with other tools, learn how to improve our tools and provide feedback to others.

The advantages of each of the four models displayed in this paper is summarized in Table I. While all the approaches are scenario based, the BP+Solver approach and the BP+DRL approach support events that have many dimensions and include numerical and discrete information. BP+Context support a disciplined way to share context data among b-threads. All three extensions have a potential to significantly simplify the models, in which respect BP+DRL seems to be most promising.

Acknoledgment: This research was supported by the Lynne and William Frankel Center for Computer Science at Ben-Gurion University.

TABLE I
Comparison of the approaches presented in this paper. BP is the baseline controller. BP+Solver is the combination of BP and the Z3 solver, BP+Context is the augmentation of BP with context aware programming and BP+DRL is the integration of deep reinforcement learning and BP.

	BP	**BP + Solver**	**BP + Context**	**BP + DRL**
Scenario based programming	✓	✓	✓	✓
Multidim. events		✓		✓
Disciplined data sharing			✓	
Potential for simplification		✓	✓	✓✓

References

[1] D. Harel, A. Marron, and G. Weiss, “Behavioral programming,” *Communications of the ACM*, vol. 55, no. 7, 2012.

[2] D. Harel, A. Marron, G. Weiss, and G. Wiener, “Behavioral programming, decentralized control, and multiple time scales,” in *Proc. of the SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pp. 171–182, 2011.

[3] A. Ashrov, A. Marron, G. Weiss, and G. Wiener, “A use-case for behavioral programming: an architecture in javascript and blockly for interactive applications with cross-cutting scenarios,” *Science of Computer Programming*, vol. 98, pp. 268–292, 2015.

[4] D. Harel and G. Katz, “Scaling-up behavioral programming: Steps from basic principles to application architectures,” in *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pp. 95–108, ACM, 2014.

[5] 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.

[6] 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*, MODELS ’18, (New York, NY, USA), pp. 59–60, ACM, 2018.

[7] D. Harel, R. Lampert, A. Marron, and G. Weiss, “Model-Checking Behavioral Programs,” in *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pp. 279–288, 2011.

[8] W. Damm and D. Harel, “LSCs: Breathing life into message sequence charts,” *Form. Methods Syst. Des.*, 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.* Springer Science & Business Media, 2003.

[10] D. Harel, A. Marron, and G. Weiss, “Programming coordinated behavior in Java,” in *ECOOP – Object-Oriented Programming* (T. D’Hondt, ed.), Springer Berlin Heidelberg, 2010.

[11] B. Shimony and I. Nikolaidis, “On coordination tools in the PicOS tuples system,” in *ICSE*, pp. 19–24, 2011.

[12] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, “Smart play-out of behavioral requirements,” in *FMCAD*, vol. 2, pp. 378–398, Springer, 2002.

[13] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss, “On composing and proving the correctness of reactive behavior,” in *EMSOFT*, pp. 1–10, 2013.

[14] H. Kugler, C. Plock, and A. Roberts, “Synthesizing biological theories,” in *CAV*, pp. 579–584, 2011.

[15] S. Maoz and Y. Sa’ar, “Counter play-out: executing unrealizable scenario-based specifications,” in *ICSE*, pp. 242–251, 2013.

[16] A. Elyasaf, A. Marron, A. Sturm, and G. Weiss, “A context-based behavioral language for iot,” *MODELS Workshops*, pp. 485–494, 2018.

[17] L. De Moura and N. Bjorner, “Z3: An Efficient SMT Solver,” in *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340, 2008.

[18] G. Katz, A. Marron, A. Sadon, and G. Weiss, “On-the-fly construction of composite events in scenario-based modeling using constraint solvers,” in *Model-Driven Engineering and Software Development, MODELSWARD 2019*, pp. 141–154, 2019.

[19] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, “Towards a better understanding of context and context-awareness,” in *International symposium on handheld and ubiquitous computing*, pp. 304–307, Springer, 1999.

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

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 2018.

[22] 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.

[23] A. Hill, A. Raffin, M. Ernestus, A. Gleave, 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.