1

Iterative ASP Pipelines: Enhancements for Robots Playing Mobile Games

Denise Angilica[1], Tayyab Ateeq[1] and Giovambattista Ianni[1]

[1] *University of Calabria*

Abstract

We overview an iterative decision making pipeline enabling a delta robot to play certain families of turn-based mobile games. The robot recognizes objects of different colors and shapes through a vision module, arranges these in the appropriate abstracted structure, and is capable of making strategic decisions based on declarative models of the game's rules and of the game playing strategy; an effector then executes moves on physical devices. The pipeline aspires to become as general and game-agnostic as possible. From this perspective, many components of the framework are designed to be reusable, including various object types and different board structures, such as square grids and others. We argue that our application provides potential for KR and robotics to be combined in creative ways, and offers itself as a general controlled environment where to experiment with forms of hybrid reasoning, while relieving from implementation details.

1. Introduction

A researcher in knowledge representation and reasoning (KRR in the following) who wants to approach robotics has to face many entry barriers: especially, one has to address many implementation details and has to attend to the tedious job of mapping quantitative and qualitative sensor and actuator data streams to the world of symbolic reasoning. We particularly focus in this paper on Answer Set Programming (ASP). ASP shows a notable potential in robotics, and indeed its capabilities in terms of declarative problem solving, diagnostic, and planning have been helpful in many remarkable robotic applications, including coordination, path finding and planning for multi-robots, arrangement and assembly of object [1]. We join the effort of leveraging ASP in robotics and enlarging the corresponding community, with the proposal

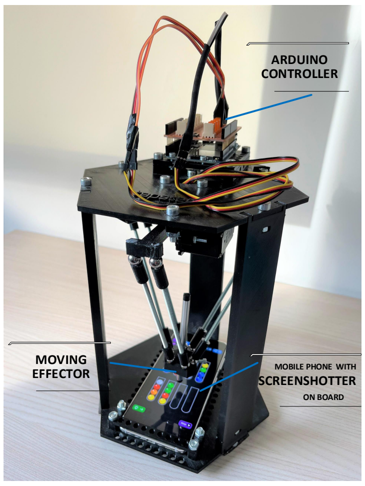

Figure 1: Main parts of the BrainyBot.

CARLA 2025: 1st Workshop on Cognitive Architectures for Robotics: LLMs and Logic in Action, September 2025, Rende, Italy.

denise.angilica@unical.it (D. Angilica); tayyab.ateeq@unical.it (T. Ateeq); ianni@unical.it (G. Ianni)
0000-0002-4069-978X (D. Angilica); 0009-0009-4180-2564 (T. Ateeq); 0000-0003-0534-6425 (G. Ianni)

of a robot-controlled and programmable environment for
experimenting and researching with mobile games. The proposed BrainyBot bridges games to robotics, eases research on the KRR aspects of these, and qualifies as an interesting testbed for neurosymbolic approaches. Designers of BrainyBot appliances are relieved from many aspects of the sensor and actuator implementation and can focus on higher-level aspects of object recognition, discretization, abstraction and reasoning. We believe our work can stimulate this line of research as it opens many possibilities such as *(i)* experimenting with new games, *(ii)* collect new benchmark data taken from actual game levels, *(iii)* investigate on practical abstraction techniques, *(iv)* explore the impact of relatively complex gestures in the reasoning workflow, such as swipes, taps, long and double taps, etc. In this paper, we briefly overview the robot's components, its run-time and design-time workflow and how it interacts with mobile phone games. We also report about recent improvements of our architecture with respect to our earlier publication [2]. The BrainyBot building instructions, its control software, example logic programs and the full implementation of games like Candy Crush Saga, Ball Sort, 2048 and others are available at https://github.com/DeMaCS-UNICAL/BrainyBot.

2. Run-time and design-time workflow of BrainyBot

The BrainyBot[1] uses a delta layout [4] for its relatively simple kinematics and ease of realization, with a single end-effector controlled by three arms moved by servo motors. The robot build includes a mobile device PH, an effector E, and a computer C (not shown in Figure 1).

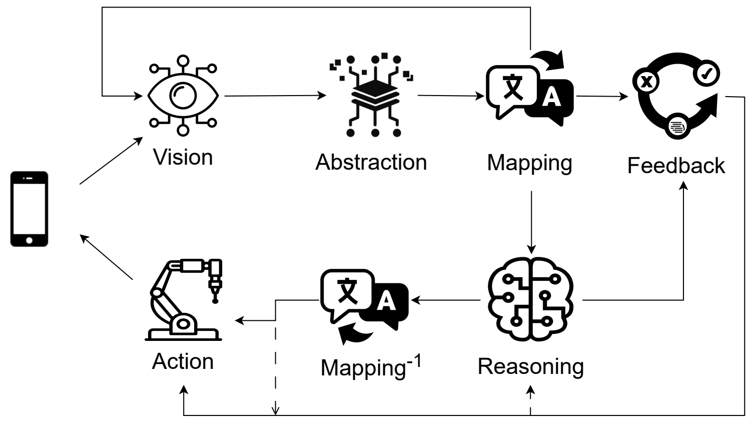

Figure 2: Run-time workflow of BrainyBot.

Run-time execution. At run-time, BrainyBot operates according to a mix of the classic sense-think-act loop and its hybrid-deliberative variant (Figure 2).

[1]BrainyBot is built according to the design of the open source TapsterBot project [3]

Sensing. A screenshot, *IM*, of what currently appears on 's display is acquired. *IM* is processed by a vision module, which recognizes relevant objects together with their raw position and other attributes like color and shape. Before sending this information to the next step, an `Abstraction` module transforms quantitative information, (e.g., pixel coordinates of objects) into an abstracted description of the current game board. This operation is performed by exploiting some game background information, like assuming that the game level contains objects forming a rectangular grid, almost default in match-3 games, or assuming that objects are laid out in stacks, a typical setting of ball sort games. Background knowledge is also helpful in filtering out false positive images, such as duplicates, or ghost detections appearing in displaced positions.

Thinking and Acting. The reasoning process begins with the game board description, obtained through the `Vision` and `Abstraction` modules, being transformed into logical assertions. These assertions, together with a declarative representation of the game rules and the selected strategy, are fed into the `Reasoning` module.

The `Reasoning` module interacts with an answer set solver S, integrated via the EmbASP framework [5], which plays the role of the `Mapping` module. The solver produces at least one optimal *answer set* A. A might encode a single move to be executed, or a sequence of moves, i.e. a plan. Furthermore A includes *feedback data*, i.e. post-conditions on the outcome of the move(s), which are necessary for validating the execution of the move/plan itself.

The execution begins with the `Action` module performing a move mv appearing in A: mv is converted back into a specific gesture to be executed on the touch display. A gesture can be either a tap or a swipe, and requires to specify proper quantitative pixel coordinates. More in detail, taps require to specify a couple of screen coordinates (x, y) where to perform a tap, whereas swipes require a 4-tuple (x_0, y_0, x_1, y_1) describing a line to be swiped while touching the screen.

The inverse and forward kinematic of a delta-robot has been widely studied since its proposal [4].

Given a desired target point $T = (x, y)$ to touch on the mobile surface, T is first converted to a tridimensional value $T' = (x_0, y_0, z_0)$ using a function $C(T, d) = T'$ where d are calibration data. It is then possible to make the robot effector reach T' by using an inverse kinematics function $F(x_0, y_0, z_0) = (\theta_1, \theta_2, \theta_3)$ where $(\theta_1, \theta_2, \theta_3)$ are the obtained target angles for the servo motors (Figure 3).

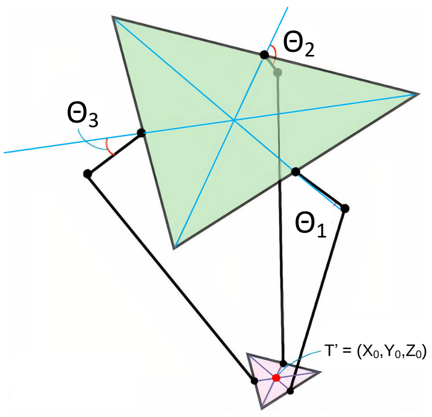

Figure 3: Abstract layout of a delta-robot.

With respect to the prototype presented in [2], two features have been introduced: *dynamic end of move detection* and *execution-feedback loop*.

Dynamic end of move detection. After executing a gesture, a typical mobile game triggers delayed effects, such as combo cascades, tube-filling animations, or other in-game transitions which alter the game board layout. BrainyBot thus necessitates to detect whether and when the game board is eventually stable in order to correctly play the next move.

In order to achieve that, multiple screenshots are captured and processed by the abstraction module; when two consecutive abstracted game-state descriptions match, the scene is assumed to be stable.

Execution-feedback loop. The execution of a move might fail or give unintended outcomes, due to physical glitches on the touch surface, bad calibration, or errors on the vision and abstraction modules. This make relevant to introduce a form of closed loop allowing to check whether a move is compliant with some specific post-conditions, and act consequently, e.g. repeating the current move, or replanning the game strategy. The checking process works as follows: once stability of the screen is detected, the system verifies that the abstraction of the current screen satisfies post-condition expectations. If this check is successful we assume that the last move produced the expected outcome.

This verification loop iterates as follows: *(i)* If the move is successful and additional moves are available in the previously computed answer set, these moves are executed sequentially, following the same "execute - wait for stable scene - verify move" cycle. *(ii)* if post-conditions are not fulfilled, the reasoning task is invoked again, generating a new sequence of moves to continue the process.

This way, the architecture ensures a continuous and dynamic feedback loop, where reasoning, execution, and verification are tightly integrated to effectively adapt to the current game state.

Design time. BrainyBot can play single-player, turn-based mobile games. Among these, BrainyBot can be programmed to play games in the Match-3 and Ball Sort family. Match-3 games are based on moves where one has to match at least three objects of the same kind in the game board, while in Ball Sort games one has to arrange colored balls put in tubes, according to a desired final goal configuration. The two families of games are good representatives of scenarios in which one has to plan in the presence of incomplete knowledge, like in match-3 games, where typically unknown objects fall in the game board from above, and with fully observable games, like ball sort games, where the game board is known in advance.

At design time, assume that a researcher R aims to use a BrainyBot for research purposes, possibly customizing its workflow for a new mobile game G. One can proceed by customizing the modules shown in Figure 2 as follows:

Screenshot and Vision: BrainyBot library functions extract screenshots from mobile phones and its Vision module recognizes fixed objects typical in Match-3 and grid-based games, colored balls as in Ball Sort puzzles, or squares containing numbers, etc. R can add new template images for arbitrary shapes and automate ball detection in screenshots.

Abstraction: R can use BrainyBot predefined modules which can reconstruct planar square and hexagonal grids for Match-3 games or object stacks for Ball Sort puzzles, which can be customized to accommodate diverse game layouts.

Mapping: BrainyBot provides predefined data structures mapped to logical predicates for modeling object grids, balls, tubes, and their relationships. The EmbASP subsystem allows R to map custom logic predicates to and from Python objects, facilitating game-specific feature modeling.

Solving: Game rules are declaratively modeled by R using ASP programs combined with game board descriptions, solved by an ASP solver to generate optimal game moves, offering flexibility for various game rules and strategies.

Acting: R can focus on high-level tasks while BrainyBot handles the execution of low-level actions.

Notes on vision and abstraction techniques. The Vision subsystem works as an abstracted upper layer on top of known computer vision methods, available in the OpenCV library, to recognize type and position of the game elements appearing in-game screenshots. Each of these techniques might be specific to the type of game at hand: a useful common technique is Template Matching (TM, in the following) which allows the recognition of fixed objects. Game objects very often have a static shape in the type of turn-based games we are targetting: we thus opted for TM as it works very well on fixed images and does not require time-consuming machine learning training and fine-tuning. For instance, in the specific case of the known Match-3 game Candy Crush Saga, all the candy object template images are stored and labeled with a type identifier that denotes the corresponding candy throughout the game. Each candidate object o has its own pixel coordinate (x_o, y_o) and a type t. The Vision module forms a collection O of candidate objects. Then the Abstraction module works by arranging elements of O in a structured arrangement g (e.g. a grid), modeled as a bidimensional matrix. Note that background knowledge about the board structure allows to eliminate possible ghost object detections.

On the other hand, objects like balls, appearing in many games, constitute a typical example of image artifact that cannot be recognized by simple template matching. Ball size, texture and color can be very different from one level to another, and it is not feasible to maintain stored template images for any possible visual appearance of this kind of object. We thus introduced a specific coordinate and color detection method for round objects. We preprocess each screenshot by converting it to grayscale, applying Canny edge detection, and extracting circular contours to locate ball centers. A Gaussian blur is then applied to smooth out texture noise, so that sampling at each center yields robust ball-color identification. Then, since in Ball Sort games balls are contained in tubes, the abstraction process will retain only circles that are contained in some tube shape. These are detected using a reference contour from a template via edge detection. We compare detected contours in the scene to this template using OpenCV's matchShapes, spotting containers despite minor scale or rotation differences. Together, these coordinate, color, and contour-matching techniques achieve reliable detection of balls and their holding containers across diverse game levels.

3. Ongoing improvements: agnostic grid detection

The vision module of BrainyBot allows for the detection of grids containing *known* repeated objects: templates of these items are stored and, while playing, the algorithm searches for their instances within the game screenshots. Therefore, the adaptation to new games is labor-intensive, due to the tedious job of retrieving templates that will appear at some point during the game. We currently are introducing some *agnostic grid detection techniques* that take place when the BrainyBot needs to be primed for a new game, and allow to automatically find and store *new unknown shapes*, later to be used during normal game-play. These techniques are based on standard image processing methods, such as the Hough Transform, a well-known technique used for finding lines and circles in an image. We are working on two different approaches:

Grid first detection. The algorithm identifies the grid structure in the image. Once the grid has been detected, objects contained within each cell are compared with each other to classify them. Each object is attached to an ID that is common to every occurrence of the same shape; two objects that are instances of two different shapes receive two distinct IDs.

Objects first detection. Objects in the scene are detected and their position is used to derive their grid arrangement; again, classification and ID assignment takes place. This integration allows the platform to generalize to new games more easily, automating a large part of the visual calibration process.

The implementation of a revised vision module, dynamically interleaving agnostic recognition methods with static template search during normal game-play, and not just when priming the robotic device for a new game, is in progress.

4. Conclusions and future work

We have enhanced the original BrainyBot prototype by introducing a number of ongoing improvements that strengthen its robustness, adaptability and ease of use:

Dynamic Feedback Loop. A continuous "sense–think-act–verify" cycle has been added: after each gesture the robot checks for scene stabilization and compares the resulting board state against the expected outcome. This verification loop —absent in the first prototype— allows immediate recovery from mis-executed taps or swipes, and seamless execution of multi-move plans without restarting vision and reasoning from scratch.

Agnostic Grid Detection. Two complementary techniques—*grid-first* and *objects-first* detection—automatically infer the underlying board structure and cluster novel shapes when priming for a new game. This reduces the manual effort of collecting and labeling templates, and lays the groundwork for completely calibration-free deployment on unseen levels.

Hybrid Vision Pipeline. We are integrating real-time interleaving of static template matching with agnostic shape recognition during regular gameplay, so that newly encountered object appearances can be learned on the fly without explicit priming steps.

Enlarged game showcase. Besides Candy Crush Saga, Ball Sort and 2048, new games are now available at the project's repository, such as Billiard and Puzzle Bobble. The first is a peculiar example of game with no arranged structure for game objects (indeed billiard balls can appear in no specific layout), while the second features hexagonal grids.

The integration of agnostic grid detection techniques is still in progress. Early experiments are promising but more extensive ones are required in order to thoroughly assess the quality of the approaches.

Acknowledgments This work was partially supported by the PNRR MUR project PE0000013-FAIR, Spoke 9 - Green-aware AI – WP9.1 and by the LAIA laboratory (part of the SILA laboratory network at University of Calabria).

Declaration on Generative AI

During the preparation of this work, the authors used Chat-GPT-4 in order to: Grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] E. Erdem, V. Patoglu, Applications of ASP in robotics, Künstliche Intell. 32 (2018) 143–149. doi:10.1007/S13218-018-0544-X.

[2] D. Angilica, M. Avolio, G. Beraldi, G. Ianni, F. Pacenza, From vision to execution: Enabling knowledge representation and reasoning in hybrid intelligent robots playing mobile games, in: P. Marquis, T. C. Son, G. Kern-Isberner (Eds.), Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023, 2023, pp. 44–54.

[3] Tapsterbot on github, Last accessed: Jun 2025. URL: https://github.com/tapsterbot/tapsterbot.

[4] P. Vischer, R. Clavel, Kinematic calibration of the parallel delta robot, Robotica 16 (1998) 207–218. doi:10.1017/S0263574798000538.

[5] F. Calimeri, D. Fuscà, S. Germano, S. Perri, J. Zangari, Fostering the use of declarative formalisms for real-world applications: The EmbASP framework, New Gener. Comput. 37 (2019) 29–65. doi:10.1007/S00354-018-0046-2.