Analyzing Action Games: A Hybrid Systems Approach

Yegeta Zeleke
University of California, Santa Cruz
Yzeleke@ucsc.edu

Joseph C. Osborn
Pomona College
joseph.osborn@pomona.edu

Ricardo G. Sanfelice
University of California, Santa Cruz
ricardo@ucsc.edu

ABSTRACT

Design support tools benefit from rich information about games' emergent behavior. Inventing successful AI players for particular games can help producing some of this information, but this is both labor intensive and limited in that it can generally only reveal that a solution exists and not say that no solution exists or that certain classes of solution exist. We show a generic method for posing and answering feasible-path, optimal-path, and reachable-space queries in action games, and we devise a measure of game level difficulty. We accomplish all this by encoding action videogame characters as hybrid dynamical systems, using Flappy Bird and Super Mario as case studies.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Formal software verification**; • **Mathematics of computing → Discrete mathematics**; **Continuous mathematics**;

KEYWORDS

Hybrid dynamical systems, Game modeling, Reachable-set, Feasible-set, Optimal-path, Game level difficulty, Control theory, Hybrid control

ACM Reference Format:
Yegeta Zeleke, Joseph C. Osborn, and Ricardo G. Sanfelice. 2019. Analyzing Action Games: A Hybrid Systems Approach. In *Proceedings of FDG (FDG'19).* ACM, New York, NY, USA, 11 pages. https://doi.org/

1 INTRODUCTION

Game design is challenging, in part, because it is difficult to anticipate the global effects of any local design change. For example, while it may be clear that adjusting the height of a platform in an action game could make that part of the level impassable, it may be less obvious that the player has an alternative route, or that this alternative route takes so much time that, although it is feasible, the player will not be able to complete the level before their clock runs out. Whether a designer is moving a platform by one pixel, tweaking a gravity or velocity constant by a small fraction to fix a bug, or changing the time duration of a powerup, there is always the possibility of opening up new (and unintended) avenues of play or closing off desirable solutions. In service of this goal, it can be helpful to have an oracle for play [19], and it can be valuable for this oracle to act as an idealized player.

If our designer's only concern were for this player to be "optimal," and if they had access to significant computational resources, they could apply techniques from reinforcement learning to obtain such an oracle for play [14]. This is an unsatisfying answer for two reasons (besides the assumption of access to a high-performance computing cluster): First, this only yields a policy for getting the most points in the least time by any means necessary, which is not necessarily the most informative thing to designers; Second, and possibly much worse, any change to the game's design—its rules, its levels, or potentially even its graphics—might require significant retraining time, depending on the architecture. In an iterative design process, this second limitation is severe. It is therefore worthwhile to look for other approaches.

In fact, hybrid systems are well-known to game developers already—but in an *ad hoc*, informal way. Designers have traditionally used informal state machine diagrams to describe game characters (a particularly readable example is Swink's treatment of Super Mario [22]), and in programming language implementations of game characters, sets of discrete variables implicitly define state machines by their combinations. For example, a Boolean variable might be set to true when a character has a speed boost powerup, which implicitly yields a normal speed state and a boosted speed state; if the character can also crawl to fit through small gaps, then we have a four-state system (crawling, with and without the speed boost, and walking normally, with and without the speed boost). The present work formalizes this combinatorial approach to game character behaviors, which admits the application of mature algorithms and approaches for analyzing such complex systems. In this paper we propose adapting a successful *hybrid dynamical systems* framework for use in games. In [15] Osborn *et al.* recently made a similar move with the related formalism of hybrid automata, and it seems expressive enough for action games as a class; open questions remained as to the computation time required to extract answers to design queries from this system. This paper builds on their results by showing that hybrid systems can also be *effective* for the purposes Osborn *et al.* established, using completely distinct solution methods with a related (but subtly different) underlying representation.

We begin by describing hybrid dynamical systems and how they differ from the hybrid automata models used by Osborn *et al.* Next, using Flappy Bird as an example, we show and solve three classes of problems related to the emergent dynamics of these hybrid dynamical systems: reachability (can the player navigate a character to a position), feasible set computation (what is the complete set of reachable positions), and optimal path planning (what is the best way, under some metric, to get a player character to a position). We showcase general techniques which are well-known in hybrid dynamical systems and which could be of significant use to developers of game design support tools. In the process, we formalize one possible metric of game difficulty in terms of the size of the intersection of the feasible set with specific points.

Note that it is exceedingly difficult to apply the reinforcement learning-based approaches mentioned above to the problem of feasible set computation. Also, while there are game-specific approaches

Research by Y. Zeleke and R. G. Sanfelice has been partially supported by the National Science Foundation under Grant no. ECS-1710621 and Grant no. CNS-1544396, by the Air Force Office of Scientific Research under Grant no. FA9550-16-1-0015, Grant no. FA9550-19-1-0053, and Grant no. FA9550-19-1-0169, and by CITRIS and the Banatao Institute at the University of California.

FDG'19, August 26-30, 2019, San Luis Obispo, CA, USA

to finding feasible sets [5, 9, 18], they leverage closed-form solutions to specific games' physical dynamics; our work only requires that the game's dynamics can be encoded as a hybrid dynamical system.

Our main contribution, therefore, is a general method which admits the use of tried-and-true solution approaches from the hybrid dynamical systems literature, including but not limited to feasibility checking, reachable set finding, and optimal path planning via model predictive control. In the future, this also opens up applications for parameter synthesis, hybrid systems identification, and other problems of potential interest in game design support tools. A secondary contribution is a possible metric of game level difficulty formalized in terms of reachable sets.

2 HYBRID SYSTEMS

Dynamical systems comprise mathematical equations describing the evolution of *state variables* defining a system with respect to time [7]. A dynamical system whose behavior is governed by differential equations is a *continuous-time system*, while a dynamical system whose behavior is dictated by a difference equation is a *discrete-time system*. A *hybrid dynamical system* (or, simply, hybrid system) is a system where its state variables can change both continuously and discretely. Such evolution of the state variable can be captured by both differential and difference equations.

Part of the definition of a hybrid system is a delineation of those regions of the state space where solutions may flow continuously (the *flow set*) and those regions where instantaneous jumps can occur (the *jump set*). A well defined hybrid system therefore consists of a differential equation, a flow set, a difference equation, and a jump set. Allowing states that can evolve both continuously and discretely, hybrid dynamical systems allow for modeling and simulation of a wide range of systems in domains including robotics, automobiles, power systems, and biological systems. Hybrid systems have been modeled and analyzed in numerous ways throughout the last few decades [4, 8, 11, 13, 23, 25].

We use the following general framework to define hybrid systems as a tuple (C, F, G, D) [8]. A closed-loop (i.e., without any input) hybrid system $\mathcal{H}$ comprises a *flow set* $C \subset \mathbb{R}^n$, which is a subset of *n-dimensional* real numbers denoted $\mathbb{R}^n$, a *jump set* $D \subset \mathbb{R}^n$, continuous dynamics given by a *flow map* $F : \mathbb{R}^n \rightrightarrows \mathbb{R}^n$, and discrete dynamics given by a *jump map* $G : \mathbb{R}^n \rightrightarrows \mathbb{R}^n$, where the symbol $\rightrightarrows$ is used to indicate that points are mapped into sets. Formally, a hybrid system is given by

$$\mathcal{H} := \begin{cases} x \in C & \dot{x} \in F(x) \\ x \in D & x^+ \in G(x) \end{cases} \tag{1}$$

To illustrate the general framework, consider a point-mass bouncing vertically on a horizontal platform. The point-mass will *flow* whenever it is rising or falling, and it will *jump* the instant it collides with the platform while falling. Suppose that, upon contact with the platform, the point-mass will immediately reverse its velocity and that a dissipation of energy will cause the speed of the point mass to reduce. Note that the mass will also flow when it is colliding with the platform but has already bounced and has an upward velocity. We can denote the *state* of the point-mass $x \in \mathbb{R}^2$ and its data as:

$$\begin{aligned} C &:= \{x \in \mathbb{R}^2 : x_1 \geq 0\}, & F(x) &:= \begin{pmatrix} x_2 \\ -\gamma \end{pmatrix} & \forall x \in C \\ D &:= \{x \in \mathbb{R}^2 : x_1 = 0, x_2 \leq 0\}, & G(x) &:= -\lambda x & \forall x \in D \end{aligned} \tag{2}$$

where x_1 is the ball's height above the platform, x_2 its vertical velocity, γ the acceleration due to gravity, and λ the restitution coefficient. When one only knows that λ belongs to the range $[\lambda_1, \lambda_2]$ with $0 \leq \lambda_1 \leq \lambda_2 \leq 1$, then the jump map G can be modeled as the set-valued map $G(x) = -[\lambda_1, \lambda_2]x$.

Solutions to the bouncing ball in (2) and to general hybrid systems $\mathcal{H}$ are parametrized by the scalar pair $(t, j) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$ where t denotes the ordinary time to represent the flow time, j stands for the number of jumps, while $\mathbb{R}_{\geq 0}$ is nonnegative real set and $\mathbb{N}$ is the set of nonnegative integers. The domain of a solution is given by a hybrid time domain, which is the collection of intervals $[t_j, t_{j+1}] \times \{j\}$ where t_j is a nondecreasing sequence denoting the jump times. See [8] for more details.

In the case of *controlled* hybrid systems, a *control input* affects the hybrid system's dynamics: the conditions determining whether a solution to a hybrid system should flow or jump are captured by subsets of the state space and the input space. Given an input[1] $(t, j) \mapsto u(t, j)$, a *state trajectory* $(t, j) \mapsto x(t, j)$ to a hybrid system has to satisfy, over intervals of flow,

$$\frac{d}{dt} x(t, j) \in F(x(t, j), u(t, j))$$

when

$$(x(t, j), u(t, j)) \in C$$

and, at jump times,

$$x(t, j + 1) \in G(x(t, j), u(t, j))$$

when

$$(x(t, j), u(t, j)) \in D$$

We can also define an *output map* of the system as a function of the system's trajectories and inputs: $y(t, j) = h(x(t, j), u(t, j))$. Finally, we can write a controlled hybrid system with state x and input u in the compact form

$$\mathcal{H}_u \; : \; \begin{cases} \dot{x} & \in & F(x, u) & (x, u) \in C \\ x^+ & \in & G(x, u) & (x, u) \in D \\ y & = & h(x, u) & \end{cases} \tag{3}$$

The objects defining the data of controlled hybrid system, are specified as $\mathcal{H}_u = (C, F, D, G, h)$. The state space x is given by the Euclidean space $\mathbb{R}^n$ while the input space is given by the closed set $\mathcal{U} \subset \mathbb{R}^m$ and the output y takes values in $\mathbb{R}^p$. Then, the set $C \subset \mathbb{R}^n \times \mathcal{U}$ contains those points in $\mathbb{R}^n \times \mathcal{U}$ on which flows are possible according to the flow map $F : \mathbb{R}^n \times \mathcal{U} \rightrightarrows \mathbb{R}^n$. The set $D \subset \mathbb{R}^n \times \mathcal{U}$ contains those points in $\mathbb{R}^n \times \mathcal{U}_d$ from which jumps are possible according to the set-valued jump map $G : \mathbb{R}^n \times \mathcal{U} \rightrightarrows \mathbb{R}^n$. See [8] and [7] for details.

3 VIDEOGAMES AS HYBRID SYSTEMS

It is challenging to predict all of the emergent behaviors possible in a videogame; this may be a part of their appeal, but it presents a challenge to game designers. Besides essential requirements of playability (can a game level be beaten?), designers must also consider softer constraints like the game's level of difficulty. These types of questions are difficult to answer without deep knowledge of the game's possible reachable states. Even relatively formal models of concepts like game difficulty have focused on abstract concepts like *rhythm groups*, structuring a level as a series of oscillations between high and low difficulty [21]. Informal approaches to game design leverage iterative changes and human testing, which is exceedingly time-consuming (and many such tests may be cut short or even invalidated if a bug is encountered during play [20]). Moreover, a simple change in the game environment can potentially impact the design and hence requires retesting and redesign.

[1] The components of u can be used to define both u_c and u_d, that is, there could be inputs that affect both flows and jumps.

In the remainder of this paper, we show how the hybrid systems framework may help answering the sorts of game design problems discussed above: Assuming that a given game can be modeled as a hybrid system, a variety of useful tools from the hybrid systems framework can be applied for analysis. First, we show that games of interest indeed can be modeled using the formalism described earlier. We then explore the application of hybrid systems techniques towards three concrete problems of interest in game design support tools: finding a witness *feasible path* that shows a particular (bad) situation is reachable or a particular (good) situation can in theory be reached; determining and visualizing the *reachable set* of system states ending in some condition (or starting from some condition), which is essentially a forall-quantified version of the first problem; and *optimal motion planning*. This latter task is commonplace in game AI, but implementations for games with continuous space and physics generally use coarse discrete approximations that are given *a priori* based on beliefs about the game's dynamics; the approach we give works on the true dynamics. We also show an application for synthesizing these different types of information: a possible measure of game level difficulty.

Example 3.1. Consider the clasical *Flappy Bird* video game, in which the player controls the vertical movement of a bird by pressing a button. Once this button is pressed, the bird moves toward the top of the screen. The game environment contains series of pipes (obstacles) that are crossing the screen horizontally at fixed speed, see Figure 1. The purpose of the game is to maintain the bird away from the moving obstacles as long as possible. The configuration of

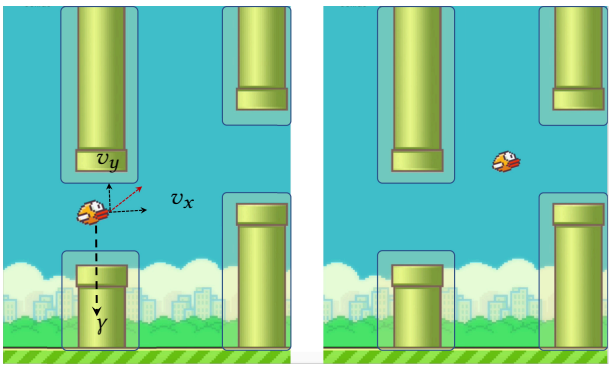

Figure 1: Flappy Bird position in a constrained environment at two time instances.

the game at each time instant is completely defined by the position, the vertical velocity of the bird on the screen, and the state of the button (pressed or not pressed). In other words, if we regard the game as a dynamical system, its solutions would describe the evolution of the bird position and vertical velocity as well as the state of the button with respect to time. Furthermore, the fact that the button has two possible values (pressed or not pressed) introduces a discrete feature to the system. For each possible configuration of the button, there is a unique continuous evolution of the system: flapping, when the button is pressed and falling, otherwise. Moreover, after each change of button configuration, a portion of the state variables representing the behavior of the bird is reset. In particular, an instantaneous change on the bird's vertical velocity occurs at button event.

The combined continuous and discrete behavior of the dynamical system $\mathcal{H}_u$ modeling this game can be seen as a hybrid system of the form (3). Let us denote by $x = (\xi, q) \in \mathbb{R}^3 \times \{0, 1\}$ the state

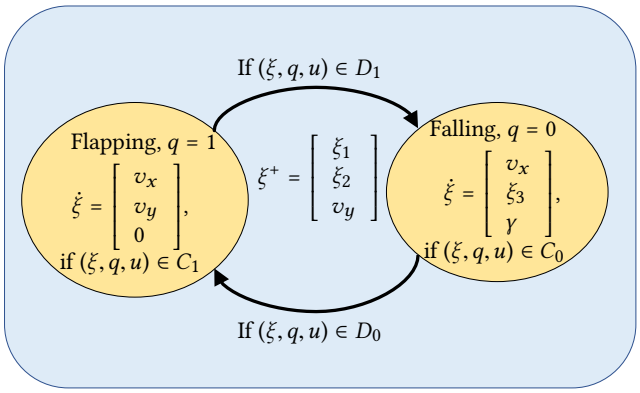

Figure 2: Hybrid automata representing the Flappy Bird game.

variable of the dynamical model, where $\xi := [\xi_1, \xi_2, \xi_3]^\top$ represents the horizontal position (ξ_1), vertical position (ξ_2), and the vertical velocity (ξ_3) of the bird, respectively, and q denotes the current mode related to button configuration. Figure 2 depicts a hybrid automata description of the continuous-time evolution (*flow*) and the discrete transitions (*jump*) of the state variables of the system. In mode $q = 1$ (*flapping*), the state ξ evolves according to the differential equation $\dot{\xi} = [v_x, v_y, 0]^\top$, where v_x and v_y are the constant vertical and horizontal velocities, respectively. In mode $q = 0$ (*falling*), the state ξ evolves according to the differential equation $\dot{\xi} = [v_x, \xi_3, \gamma]^\top$ where γ is a constant acceleration due to gravity. A switch between the modes is caused by an external action (player) which can be considered as an input to the system and denoted by $u \in \{\text{ON,OFF}\}$. In order to completely define the game as a hybrid system, we shall specify the region of the augmented space composed of state and input spaces where the *flow* occurs. This region is denoted by $(x, u) \in C$. Similarly, we introduce the region of the same augmented space where the *jump* takes place which we denote by $(x, u) \in D$.

A transition from mode $q = 1$ to mode $q = 0$ (respectively from mode $q = 0$ to mode $q = 1$) occurs when $(\xi, q, u) \in D_1 = \mathbb{R}^3 \times \{1\} \times \{\text{OFF}\}$ (respectively $(\xi, q, u) \in D_0 = \mathbb{R}^3 \times \{0\} \times \{\text{ON}\}$). During this transition, ξ_1 and ξ_2 stay the same, however, ξ_3 resets to v_y which is the initial falling speed. In mode $q = 1$ (respectively mode $q = 0$) the system will be flowing as long as $(\xi, q, u) \in C_1 = \mathbb{R}^3 \times \{1\} \times \{\text{ON}\}$ (respectively $(\xi, q, u) \in C_0 = \mathbb{R}^3 \times \{0\} \times \{\text{OFF}\}$).

The Flappy Bird game is completely defined by the following hybrid system with only one input affecting the flow and jump sets:

$$\mathcal{H}_u^{FB} : \begin{cases} \dot{x} = F(x) & (x, u) \in C := \bigcup_{q \in \{0,1\}} C_q \\ x^+ = G(x) & (x, u) \in D := \bigcup_{q \in \{0,1\}} D_q \end{cases} \tag{4}$$

where

$$x = (\xi, q), \quad \begin{cases} C_0 = \mathbb{R}^3 \times \{0\} \times \{\text{OFF}\} & D_0 = \mathbb{R}^3 \times \{0\} \times \{\text{ON}\} \\ C_1 = \mathbb{R}^3 \times \{1\} \times \{\text{ON}\} & D_1 = \mathbb{R}^3 \times \{1\} \times \{\text{OFF}\} \end{cases}$$

and

$$F(x) = \begin{cases} \begin{pmatrix} v_x \\ v_y \\ 0 \\ 0 \end{pmatrix} & \text{when } q = 1 \\ & \forall x = (\xi, q) : (x, u) \in C \\ \begin{pmatrix} v_x \\ \xi_3 \\ -\gamma \\ 0 \end{pmatrix} & \text{when } q = 0, \end{cases}$$

$$G(x) = \begin{pmatrix} \xi_1 \\ \xi_2 \\ v_y \\ 1 - q \end{pmatrix} \qquad \forall x : (x, u) \in D$$

Example 3.2. *Flappy Bird* has an interesting property which is consistent with many standard problems in motion planning: every obstacle is to be avoided. This is mainly because the bird dies immediately on any collision; but there are plenty of games that do not have this property. Consider, for example, *Super Mario World* (Figure 3). In this game, the player-controlled character must be on the ground in order to run or jump, and in many cases must hit its head against a breakable ceiling in order to make progress. Since including an environment which the player can modify would force us to consider task planning in addition to motion planning, for now we consider a constrained version of the Mario character who cannot break any blocks. Even so, it provides an interesting generalization.

Figure 3: Environment from *Super Mario World*. The controlled character may change size, and some obstacles act as floors but not ceilings.

Super Mario World has a variety of different powerups and additional abilities, but to simplify the presentation we concern ourselves here with the constrained version of Mario who is always the same size and has no abilities besides running, standing still, and jumping. We also consider only environments with obstacles that are always floors, walls, or ceilings (like the yellow blocks in Figure 3), although this constraint is easily relaxed by considering the character's instantaneous state (e.g., whether the character is ascending or descending). Finally, we exclude the world map from consideration here and address only individual stage environments.

A configuration of our simplified *Super Mario World* therefore includes the continuous position and velocity of the player character and other on-screen (and potentially off-screen) entities as well as discrete variables including the states of control inputs. Again, treating this game as a dynamical system we can see that the solutions to this system describe the evolution of Mario's position and velocity. The discrete features here include whether the left or right directional input (joystick's directional button) is given and whether the jump button is pushed (held) down.

This induces two modes of operation, one for each combination of inputs: a) direction of motion and b) jump button. However, like Flappy Bird, Mario has some unusual properties. For instance, the acceleration due to gravity while falling (i.e., after the apex of his jump) is *higher* than it is while Mario is rising. Even stranger, the longer the jumping button is hold (up to a certain threshold), the higher Mario jumps! So, after the jump is initiated the controller can still determine how high the jump must be. Mario can also accelerate horizontally while in the air, although at a reduced acceleration. Another consideration is that Mario has maximum and minimum X and Y velocities, essentially requiring terminal-velocity states. The discrete state space of Mario is therefore much larger than Flappy's.

Similar to the case-study of Flappy Bird, the combined continuous and discrete behavior of the dynamical system modeling this game can be seen as a hybrid system of the form (3). Let us denote by $x = (\xi, q, \tau) \in \mathbb{R}^3 \times \{0, 1\} \times \mathbb{R}_{\geq 0}$ to represent the state variable of the dynamical model, where $\xi = [\xi_1, \xi_2, \xi_3]^\top$ contains the horizontal position (ξ_1), vertical position (ξ_2), and the vertical velocity (ξ_3) of Mario, respectively. The state q indicates the discrete variable in Mario's dynamical model (when Mario is on the air or in contact with the ground), and τ denotes the amount of time the jump input is provided. The system's input is given by $u = (u_1, u_2) \in \{-1, 0, 1\} \times \{0, 1\}$, where u_1 denotes the current directional control (left, stay, or right, respectively) u_2 denotes the current jump button configuration (button pushed or not pushed, respectively).

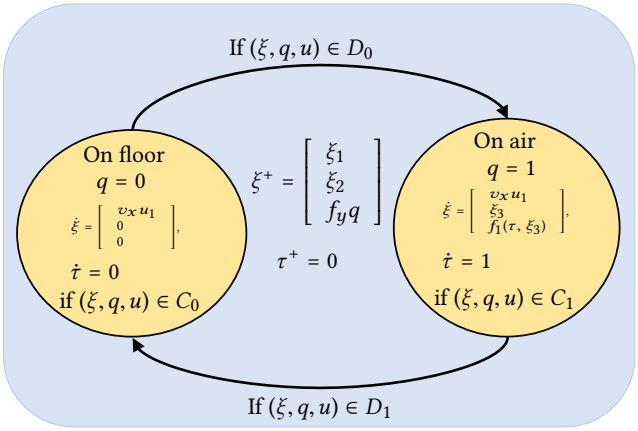

Figure 4: Hybrid automata modeling Super Mario World.

Figure 4 depicts a hybrid automata describing both the continuous-time evolution (*flow*) and the discrete transitions (*jump*) of the state variables of the system. The logic it implements is as follows:

- In mode $q = 0$, Mario is in contact with the ground or standing on a floating surface which is either standing still, running right or running left, depending on u_1. Hence, the state evolution is given by
$$\dot{x} = [v_x u_1, 0, 0, 0, 0]^\top$$
where v_x is the horizontal velocity.
- In mode $q = 1$, Mario jumps right (if $u_1 = 1$), left (if $u_1 = -1$), or straight up (if $u_1 = 0$) allowing the state x to evolve according to the differential equation
$$\dot{x} = [v_x u_1, \xi_3, f_1(\tau, \xi_3), 0, 1]^\top$$
where the function
$$f_1(\tau, \xi_3) := \begin{cases} -\gamma_{\text{down}} & \text{when } \xi_3 < 0 \\ \gamma_{\text{up}} + \alpha(\tau) & \text{when } \xi_3 \geq 0, \end{cases} \tag{5}$$
denotes vertical acceleration during Mario's jumps, while the term $\alpha(\tau)$ is used to indicate the longer the jump input is active, the smaller the deceleration rate is used; thus, the higher the jump will be. Therefore, one can define $\alpha(\tau) = a\tau$ where a is a positive constant and τ is used to denote the amount of time the input u_1 is held. The positive constants γ_{up} and γ_{down} represent the rate of acceleration when Mario is *rising* and *falling*, respectively.
- A switch between the modes is caused by an external player action changing the input u. In this sense, a transition from mode $q = 0$ to mode $q = 1$ occurs when $(\xi, q, u) \in D_0 := \mathbb{R}^3 \times \{0\} \times \{1, 0, -1\} \times \{1\}$. Upon this transition, the state is reset according to the difference equation
$$x^+ = [\xi_1, \xi_2, f_y q, 1 - q, 0]^\top$$
where f_y is the initial vertical velocity and $1 - q$ determines the next mode. These transitions are necessary to show the character Mario is now on the air (either jumping up or falling down) or on the ground as it is dictated by the corresponding differential equations.

To account when the character returns to ground after a jump, we introduce the jump set
$$D_1 := \mathbb{R} \times \mathcal{B} \times \mathbb{R} \times \{1\} \times \{-1, 0, 1\}$$
where $\mathcal{B}$ is defined as a set that indicates when the player character is in contact with a floor. The overall conditions for jump can be defined as $D := \bigcup_{q \in \{0,1\}} D_q$. Furthermore, the flow set is $C := \bigcup_{q \in \{0,1\}} C_q$ where $C_0 := \mathbb{R}^3 \times \{0\} \times \{-1, 0, 1\} \times \{0\}$ and $C_1 := \mathbb{R}^3 \times \{1\} \times \{-1, 0, 1\} \times \{1\}$. Consequently, we introduce Mario's dynamics as the following hybrid system:

$$\mathcal{H}^M := \begin{cases} \dot{x} = F(x, u) & (x, u) \in C = \bigcup_{q \in \{0,1\}} C_q \\ x^+ = G(x, u) & (x, u) \in D = \bigcup_{q \in \{0,1\}} D_q \end{cases} \tag{6}$$

where

$$F(x, u) := \begin{cases} \begin{pmatrix} v_x u_1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} & \text{when } q = 0, \\ \begin{pmatrix} v_x u_1 \\ \xi_3 \\ f_1(\tau, \xi_3) \\ 0 \\ 1 \end{pmatrix} & \text{when } q = 1, \end{cases} \qquad \forall (x, u) \in C$$

$$G(x) := \begin{pmatrix} \xi_1 \\ \xi_2 \\ f_y q \\ 1 - q \\ 0 \end{pmatrix} \qquad \forall x : (x, u) \in D$$

4 REACHABILITY AND FEASIBILTY

Among the most important challenges in video game analysis are quantifying the playability of the game within the player's ability and gauging the difficulty of the game with respect to changes in the game environment. Solving the later challenges would considerably help the game design support system.

For games modeled as hybrid dynamical systems, one can utilize existing approaches in order to handle the aforementioned challenges. Indeed, given a hybrid system model whose solutions describe the game character's evolution with respect to time, the player's action on the game represents control inputs for the game's dynamical model. As a result, the solution to game model must at least ensure that the character never interpenetrates the environment inappropriately. If a target point—a final destination for the character—is given, then the system's solutions must also end up in the target point or area. Analyzing playability therefore consists in providing answers to two main questions: First, what is the *set of initial configurations* of the character that allow for an *admissible input* such that the character trajectory reaches the target? Second, can we quantify the difficulty of reaching the target from the obtained set of feasible initial points? And if so, can we use this information to provide assistance to game designers?

It turns out that the hybrid systems literature offers powerful approaches for addressing these questions. Solving a feasibility problem for a given target set consists of analyzing the reachable set from the target set in backward time. Indeed, this backwards-reachable set contains the complete admissible set of initial points and can also characterizes the difficulty of reaching a target point in forward time by providing all the possible solutions that allow the character to reach the target point.

In the remainder of this section, we study a reachability problem for hybrid systems and answer the previously stated questions by studying a feasibility problem for hybrid systems. Specifically, we propose an algorithm to approximate the set of reachable points from a given initial point.

4.1 Reachability

Reachability analysis consists in proposing methods and algorithms to approximate the set of points, the so-called reachable set, generated by the system solutions starting from a given set of initial points, denoted χ_o, after finite (hybrid) time (T, J). More precisely, the reachable set from χ_o up to (T, J) for $\mathcal{H}$ is given as

$$\text{reach}^{\mathcal{H}}_{(T,J)}(\chi_o) := \{x(t, j) : x(0, 0) \in \chi_0,\ t \leq T,\ j \leq J,\ (t, j) \in \text{dom}\, x\} \tag{7}$$

where x is a solution to $\mathcal{H}$ from χ_o.

Since the game solution lives in a constrained environment, we are more interested in analyzing safe reachable sets rather than reachable sets. That is, the safe reachable set is given by $\text{reachsafe}^{\mathcal{H}}_{(T,J)}(\chi_o) := \{x(t, j) : x(0, 0) \in \chi_0, x(t, j) \notin \chi_u \forall (t, j) \in \text{dom}\, x : t \leq T, j \leq J\}$ where χ_u is used to denote the unsafe set (e.g., obstacles region).

Reachability analysis is a key step in the verification of hybrid systems. The set $\text{reachsafe}_{(T,J)}(\chi_o)$ indicates whether solutions

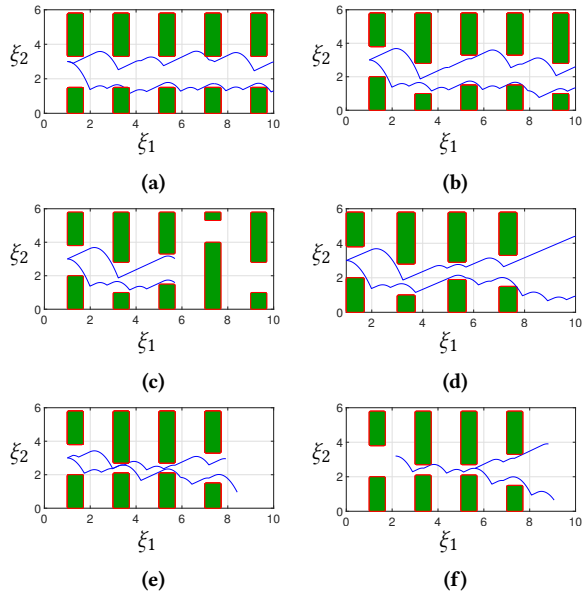

Figure 5: Simulation result depicting safe reachable set for different obstacle configuration.

originating from a given initial condition χ_0 satisfies the constraints (avoiding χ_u) or not. Given a controlled hybrid system of the form (3) and the unsafe set χ_u, we are interested in finding the set of input signals $u : \text{dom } u \to \mathbb{R}^m$ such that the resulting solutions belong to $\text{reachsafe}^{\mathcal{H}}_{(T,J)}(\chi_0)$ for a given $(T, J) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$. Many reachability analysis algorithms have been proposed in the literature, including [3, 6, 24].

We consider Example 3.1 to illustrate a way to compute the set $\text{reachsafe}^{\mathcal{H}}_{(T,J)}(\chi_o)$.[2] In the context of this example, the bird has to stay in a safe set before (T, J). We first discretize the hybrid system $\mathcal{H}^{FB}_u$ in (4) with a fixed step size using the Hybrid Equations Toolbox package (HyEQ) [16]. Since the range of the system's inputs is discrete and finite ($u \in \{0, 1\}$), there exist 2^N possible sequence of inputs, where N is the total number of time steps in the interval $[0, T]$. The resulting discretized solution is denoted by $x_{\{0,1,\ldots,N\}}$.

For the obstacle setup shown in Figure 5, computing the admissible inputs can be achieved using iterative methods, e.g. Algorithm 1. The algorithm first computes the upper bound of the safe solutions (line 3-10). That is, it computes the solutions of (4) from the initial condition χ_0 and checks if the constraints are met for all possible combination of inputs. Similarly, it computes the lower bound of the safe solution (line 12-19).

In Algorithm 1, we first compute the upper-bound trajectory of the reachable set (lines 2-11), then we compute the lower-bound trajectory (lines 12-20). Indeed, the 2^N input sequences are ordered to increase with respect to the index i ($i = 0$ corresponds the sequence of all-zero inputs and $i = 2^N + 1$ corresponds to the sequence of all-one inputs). `getNextInput(i)` generates the i-th sequence of inputs. That is, the first loop iteration (lines 3-11) starts from all-one bit stream and tries to find the first sequence of inputs that satisfies the constraints. Once the first bit stream that satisfies the constraints is found (line 6), the loop breaks and

[2]Source code available at https://github.com/HybridSystemsLab/FlappyBirdReachability

Algorithm 1: Computation of reachable safe set. reachsafeComputation (χ_u)

```
1  Obtain initial system state set χ0 and the horizon N
2  i = 0
3  while i < 2^N do
4      u-upper_{0,1,...,N} = getNextInput(i)
5      Simulate system (4) from χ0 for input signal
         u-upper_{0,1,...,N} to obtain x_{0,1,...,N}
6      if x_{0,1,...,N} ∉ χu then
7          upperBound=True
8          break
9      end
10     i = i + 1
11 end
12 i = N
13 while i > 0 and upperBound do
14     u-lower_{0,1,...,N} = getNextInput(i)
15     Simulate system (4) for input signal u-lower_{0,1,...,N} to
         obtain x_{0,1,...,N}
16     if x_{0,1,...,N} ∉ χu then
17         break
18     end
19     i = i − 1
20 end
```

the algorithm proceeds to find the lower-bound trajectory for the reachable set in a similar way (lines 12-20). In the case where all the input sequences are exhausted without attaining a viable solution ($\text{reachsafe}^{\mathcal{H}}_{(T,J)}(\chi_0) = \emptyset$), the upper-bound trajectory will be empty and hence line 13-20 will not be executed.

In Figure 5, the blue trajectories indicate the lower-bound and the upper-bound trajectories of the safe reachable set. Given a point in $\text{reachsafe}^{\mathcal{H}}_{(T,J)}(\chi_o)$ within the blue inscribed region, there exist a sequence of inputs such that the corresponding solution reaches x when starting from χ_o. This conclusion is highly relevant to design support tools! Specifically, knowing the reachable set and its boundaries can help with placing rewards or intractable objects.

4.2 Feasibility

In this section, we exploit the tools developed in Section 4.1 in order to address the feasibility problem. The objective of the feasibility problem is to compute a set of initial points χ_0 such that the solutions starting from it reach a given target set denoted χ_F. Recall that in the reachability problem we compute the set of final points χ_F reached by the solutions starting from a given set of initial points χ_0. Our approach relies on the following key observation: the set of initial points χ_0 obtained from solving the feasibility problem is a set of final points reached by the backward solutions starting from the given target set χ_F.

Motivated by this, we compute the backward-in-time solutions of the given system $\mathcal{H}_u$ to solve the feasibility problem for the (forward-in-time) system $\mathcal{H}_u$ as follows: we obtain the final set χ_F for the feasibility problem for $\mathcal{H}_u$ by solving backward-in-time reachability problem with χ_F as the initial set. For this purpose, we define a backward-in-time version of $\mathcal{H}_u$ and then compute the solutions to that system forward in time.

Before analyzing the backward-in-time reachability problem (namely, the feasibility for $\mathcal{H}_u$) from the set χ_F, it is important to notice that for a high order flow model, the set χ_F may not be given in terms of all the state variables involved in the modeling. In this case, it is important to find a way, before computing the system's backward solutions, to augment the set χ_F so as to specify all of the state variables. For example, if we consider the Flappy Bird model introduced in (4), we notice that the state contains the bird position and vertical speed. However, in practice, the target set χ_F corresponds to only the character's target positions (the character's final velocity is not specified). Therefore, before computing the backward solutions one has to compute first the possible character's final velocities. One natural method to do this consists in computing (offline) all possible final velocities for any possible sequence of inputs along a given time horizon. This method can be used for any high order character's dynamics provided that the control inputs are assumed to take finite and discrete values. In our approach for Flappy Bird, the computed final velocity (or remaining state variables for general high order flow dynamics) corresponding to each sequence of inputs along the given time horizon, are gathered in a look-up table and serve to initialize the backward solutions.

So far, we have proposed a general approach to compute the missing state variables in the specified final state χ_F. However for the Flappy Bird model, where the inputs are discrete and finite ($u \in \{0, 1\}$), the final velocity (initial velocity for the backward system) can be computed in a simpler way while observing that character's velocities when flapping are constant. Indeed, starting from a point on the backward solution and given a sequence of inputs, by counting the number of consecutive backward time steps along which the system is falling (mode $q = 0$) and solving the differential equation $\dot{v_f} = 0$ with $v_f(0) = v_y$ along all consecutive falling time steps, the obtained velocity v_f at the end of the consecutive falling steps is an admissible final velocity (initial velocity for the backward system) for the considered sequence of inputs. More specifically, to initialize the backward speed we use a *key-value* lookup table. That is, we compute the initial backward velocity using the forward dynamics after each l time steps, with $l = 1, 2, ..., N$, and record the obtained result in the table with a $key = l$ and $value = v_f$, where v_f is the final velocity after falling l time steps. In this way, for feasibility problem with N time steps, the look up table will have N entries.[3] Since this lookup table will be computed offline, the computational burden is minimal and the access time is constant.

For various obstacle setups shown in Figure 6 (a-d), the obtained safe backward solutions are plotted starting from χ_F along N time steps where N denotes the time horizon (or number of time steps) and χ_F is the final set.[4] In general, it is very time consuming to compute all the backward solutions starting from χ_F especially when the horizon N is large. One way to handle this inadequacy consists in solving the feasability along consecutive sub-horizons. Indeed, starting from χ_F we compute all the feasible backward solutions along a given first sub-horizon. At the end of the sub-horizon we obtain a set of points that are reached. From the later set, we select only a subset of points from which we re-solve the feasability problem along another backward-time sub-horizon. We iteratively apply the same approach until reaching the region where the character is supposed to start.

[3] An alternative way to perform such computation is to define a backward-in-time version of $\mathcal{H}_u$ and compute the solutions to it forward in time. Such solutions are backward-in-time solutions to $\mathcal{H}_u$.

[4] Source code available at https://github.com/HybridSystemsLab/FlappyBirdFeasibility

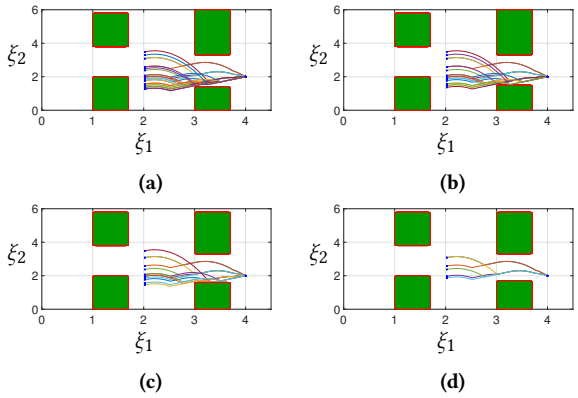

Figure 6: Simulation result depicting feasible initial sets for different obstacle setup using Algorithm 2 at first sub-horizon.

Algorithm 2: Computation of feasible set feasibleSet (χ_F, k, ξ_{Target}, N_{sub})

```
1  if ξ1 = ξ_Target then
2  |  return (True)
3  end
4  validInputs= getInputSequence(N_sub, χ_F)
5  if isempty (validInputs) then
6  |  return (False)
7  end
8  for i ∈ {0, 1, ..., k} do
9  |  x_terminal = getTerminalPoints(χ_F,validInput(i))
10 |  found = feasibileSet(x_terminal, k, ξ_Target, N_sub)
11 |  if found then
12 |  |  save data
13 |  |  plot
14 |  end
15 end
```

In Algorithm 2, we solve the feasibility problem using a recursive subroutine. The algorithm takes in four parameters, namely χ_F, k, ξ_{Target}, and $N_{sub}$to indicate the target point, the number of inputs that are selected and propagated along the next sub-horizon, the targeted horizontal axis to stop the recurrence, and the sub-horizon $N_{sub} \leq N$, respectively. The function getInputSequence() takes in N_{sub} and the initial condition for the backward-in-time system, which is χ_F and gives a list of valid input sequences from χ_F (line 4). Once all valid inputs are determined the next step is to apply selected valid input sequences and obtain terminal points to be used as initial condition for next recurrence (line 10).

The remaining question is how many terminal points do we consider as an initial point to propagate backward solutions Therefore, we provide a flexible method by allowing the user pick k, the number of points in the feasibility set to be propagated backward. In Figure 6, $k = 256$ for $N = 8$ is used to calculate the backward-in-time solutions. Indeed this induces more computational burden if we propagate backward more than one set, however, for computing only the first set, the computational burden is mild. Experiments

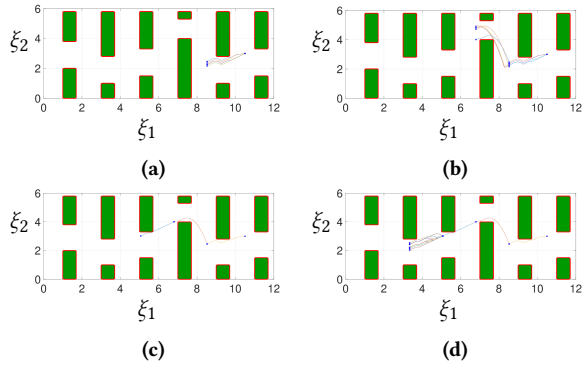

Figure 7: Simulation result depicting feasible sets at different sub-horizon using Algorithm 2.

were conducted on a personal computer powered by an Intel core i5 processor with 2.7 GHz clock speed and 8GB RAM. For the feasible sets shown in Figure 6 (a-d), the computation time is 42.6 s, 10.9 s, 5.5 s, and 7.5 s, respectively. Similarly, solving the feasibility problem shown in Figures 7 (a-d) and 8 (a & b) took 5.8s, 35.7 s, 85.02 s, 216.2 s, 0.33 s, and 0.2 ms respectively.

In Figure 7, the feasibility problem is solved for various obstacle setups. In particular, we explore how the feasibility set get affected in relation to changes in the game environment, namely the obstacle setup. Indeed, as the gap between obstacles get smaller and smaller, the feasible set shrinks. This particular behavior provides an essential technique for game design. More specifically, one can gauge the difficulty of the game by just looking at the solution of the feasibility problem.

Consider the game setup shown in Figure 7 (a)-(d). In the figures, a sub-horizon of $N_{sub} = 8$ and $\xi_{Target} = 3.2$ with $k = 2$ is used while the target point is set to $\chi_F = (\xi_1, \xi_2) = (10.5,\ 3)$. That is, after a sub-horizon (N_{sub}), maximum or 23 terminal points are selected from all terminal points in the set and propagated backward-in-time on the next sub-horizon. In Figure 7(a) (respectively in (b)), the feasibility problem is solved to obtain the first(respectively the second) set backward-in-time. As shown in the figure, there exist multiple solutions for the feasibility problem both in (a) and (b). That is, for the particular target point chosen, the bird's position can start at various point within the feasible set. However, this is not the case for Figure 7 (c), where the solution is extended for one more set backward-in-time. That is, the number of possible trajectories drastically diminishes as the obstacle setup in $\xi_1 \in (5, 5.5)$ hinders many trajectories from passing through. In (d), the solution expands to include various initial condition as the obstacle setup relaxes. Although the solution expands at the later sets, it is important to notice the solution will have one single path when $\xi_1 \in (5, 7)$. This yield in increased difficulty in playing the game as the player has to get this exact solution. In most games, it is desirable the player has enough possibilities in choosing the safe path, and our numerical results show that this is not the case for the parameter used.

Now, consider the feasibility solution shown in Figure 8. In sub-figure (a), the initial condition $\chi_F = (10.8, 3.5)$ is used. It can be seen from the sub-figure that there exists only one solution that can lead the trajectory of Flappy Bird starting at $\xi_1 = 7$ towards the desired destination given by χ_F . In sub-figure (b), the initial condition was

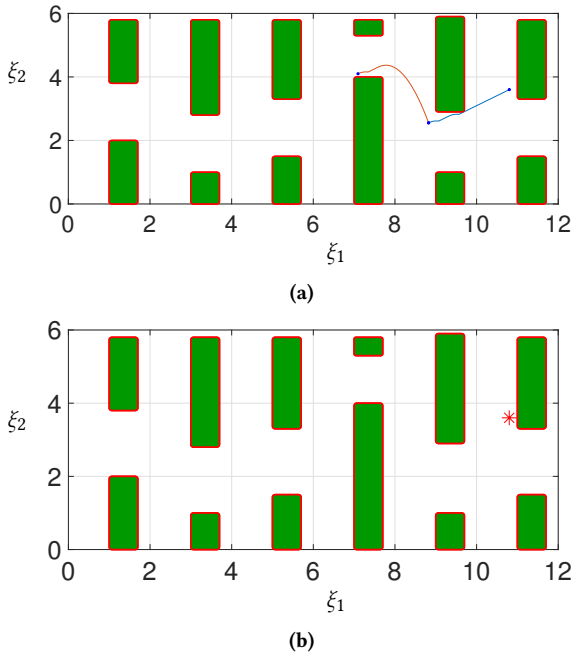

Figure 8: Simulation result showing unique solution and no solution for selected initial conditions

set to be a little higher at $\chi_F = (10.8, 3.6)$ (Marked in red $\star$). One can notice the solution for the feasibility problem is the empty set. That is, no matter where the initial condition along the vertical line $\xi_1 = 7$ and $\xi_2 \in (0, 6)$ (the playable vertical area), there exist no input sequence that guides the bird to the target point (marked as green star).

In order to analyze the playability of the game it is possible to introduce a metric to quantify the difficulty level of the game or even the existence of solution for a specific game. One possible approach for gauging the difficulty consist in introducing the following min-max cost function:

$$J(\chi_F, T) := \min_{t \in [1,T]} \max_{(a,b) \in \text{reachsafe}_t^{b,\mathcal{H}_u}(\chi_F)} |a - b|^2. \tag{8}$$

where $\text{reachsafe}_t^{b,\mathcal{H}_u}$ is the backward reachable set with safety up to ordinary time t. The function J is equal to zero when the feasible solutions are unique along some intervals, and when J is large, it follows that the target point can be reached through different paths. Hence, the player will have a large freedom to win the game. For the particular game setup shown in Figure 7 (a)-(f), $J(\chi_F, T)$ is 0.2, 0.22, 0, 0, 0, and 0, respectively. Therefore, the difficulty of the game is inversely related to the cost function J.

5 PATH PLANNING USING MPC

Thus far, we have proposed approaches to calculate all possible input sequences that take the game character or the system's trajectories to a desired target set/point while avoiding the obstacles. Now, we propose methods to select the optimal input sequences among all the feasible ones with regards to an optimality criteria. The optimality criteria can be chosen so as to select input sequences that allow fast convergence to a goal or to require a minimum number of jumps, or time to accomplish a task (equivalently, leading to

a minimum number of actions required from the player). Randomized sampling-based planning algorithms such as RRT have been widely used in motion planning problems, however, RRT converges to sub-optimal solution [10]. Informed searching algorithms such as A* are complete and optimal, but obtaining a heuristic function that guides the search is challenging. Moreover, in most planning algorithms complex dynamics are not considered, rather the trivial point mass dynamical model is utilized. One can track the result obtained from those algorithms (i.e. RRT, A*,D*...) to tackle path planning problem, however guaranteeing safety can be compromised since the dynamical constraints are not considered in the planning process.

5.1 Outline of Approach

We start by formulating the motion planning problem for a hybrid system $\mathcal{H}_u$.

Problem ($\star$): Given a hybrid system $\mathcal{H}_u$ modeled as (3) with initial state x_0 and final state x_F, determine the input

$$(t, j) \mapsto u(t, j) \tag{9}$$

such that resulting solution x to $\mathcal{H}_u$ in (3) and from

$$x(0, 0) = x_0$$

is such that for some $(T, J) \in \operatorname{dom} x$,

$$x(T, J) = x_F$$

In order to tackle Problem ($\star$), we propose a general hybrid MPC framework. Motion planing is a fundamental component in video games testing and it can be implemented in static, dynamic, and real-time environments. Numerous developments have improved the accuracy and effectiveness of motion planing techniques over the past two decades. The core of motion planning relies heavily in provision of high-performance feasible path generation whether be a single-agent or a multi-agent system. The presence of dynamic changes in the environment, heterogeneous terrains, and variable rewards make solving motion planing problem difficult. In particular, planning a path such that all constraints are satisfied and a certain feature is maximized could pose a difficult problem.

Though the rules of Flappy Bird are simple to understand and executed by a player, it poses some difficulties for an automated controller. The combined continuous and discrete behavior of the system as well as the constrained nature of the state space due to the existing time-varying obstacles makes the application of classical control techniques not very useful [17]. One possible approach to handle this shortcoming consists in decomposing the problem into two independent sub-problems: first, we generate a reference trajectory that considers only the presence of the obstacles offline, then, once a reference trajectory is generated, solve the *trajectory-tracking* sub-problem by designing a control law that allows the actual to converge asymptotically to the reference trajectory. Decoupling the problem into sub-problems has the advantage of reducing complexity. Indeed, the complexity rising from the nature of the obstacle is not considered in the planning phase. Similarly the geometric constraint such as obstacle location and boundaries are not considered in the trajectory-tracking phase. Unfortunately, this method could result in an unreachable reference trajectory (namely, a trajectory that can not be attained due to the system dynamics) since dynamical restrictions are not taken into account at time of motion planning [12]. To handle this issue, we propose safe real-time planning approach in constrained environments while considering the hybrid nature of the system's dynamics. The proposed approach is introduced in the next section. It utilizes the ideas in Section 4.

5.2 MPC Formulation for Planning

This section formulates an optimization problem whose solution defines a reference trajectory while considering both dynamical and geometric constraints of the game. In particular, we are interested in solving the following optimization problem using a predictive hybrid control scheme [1].

Problem 5.1. Given $T \geq 0$ and $J \in \mathbb{N}$ defining a prediction horizon, terminal cost V, unsafe set χ_u, functions L_c and L_d, find a hybrid arc $x^\star$ and a hybrid input $u^\star$ with compact hybrid time domain $\operatorname{dom} x^* = \operatorname{dom} u^*$ and solution to $\mathcal{H}_u$ that minimizes

$$\begin{aligned} \mathcal{J}(x, u) \quad := \quad & \sum_{j=0}^{J} \int_{t_j}^{\min\{t_{j+1}, T\}} L_c(x(t, j), u(t, j))dt \\ & + \sum_{j=0}^{J-1} L_d(x(t_{j+1}, j), u(t_{j+1}, j)) + V(x(T, J)) \end{aligned} \tag{10}$$

subject to

$$x(t, j) \notin \chi_u \qquad \forall (t, j) \in \operatorname{dom} x, \qquad (T, J) \in \operatorname{dom} x$$

where x denotes a solution to $\mathcal{H}_u$ with input u. The function L_c determines the cost of flow and is defined on the flow set C. The function L_d is defined on the jump set D and determines the cost of jumps. The function V denotes the terminal cost. The functional $\mathcal{J}$ in (10) combines the flow and jump costs along with the terminal constraint.

We utilize a Model Based Predictive Control (MPC) [1] [2] scheme for hybrid dynamical systems $\mathcal{H}_u$ to obtain solutions given as sequences of inputs. That is, we predict sequences of points $x_{\{0,1,\ldots,N\}}$ according to the system dynamics in (3) and possible inputs. Then we select the sequence with the least cost. This is done by solving Problem 5.1 for a given prediction horizon (N) to obtain the optimal input sequence $u^\star$ and applying it for the duration of the control horizon (M). The terminal state obtained after applying $u^\star$ for M time steps will then be used as the initial condition for the next horizons and the process is repeated. The algorithm computing the optimal input sequence is outlined below. In the algorithm, $\mathcal{H}_{u_i}$ denotes input u for system $\mathcal{H}_u$ at the i-th iteration.

Algorithm 3: MPC(x_0, x_F, M)

```
while x_0 ≠ x_F do
    Solve Problem 5.1 to get input u_{0,1,...,M}
    Discretize the hybrid system H_u and simulate using the
      HyEQ toolbox to obtain x_{0,1,...,M}
    Set x_0 = x_{M}
end
```

In conventional MPC, the existence of solution often depends on the initial input guess. Indeed, the solution to the hybrid system, which is solved in MPC will critically depend on the initial input guess. To illustrate this, we consider the reachable set presented in Figure 5(e). For a choice of initial guess $u = 1$ (flapping), there exists no solution. However, for the "right" initial guess there is nonunique solution that leads the bird trajectory away from obstacles for a specified time horizon and initial condition.

Example 5.1. Suppose our goal is to produce the smoothest safe solution possible. In this case, one would like to minimize the amount of state transition (push buttons) in the game. We follow the general formulation stated in (10) and set $L_c = 0$ while defining $L_d \in \{0, 1\}$

to indicate if a jump happens at time $t_j \in [0, T]$. That is, the function L_d returns the number of jumps in the interval $t \in [0, T]$ for a given solution. We are interested in designing a control law u that complies with (4) while guarantying minimal cost $\mathcal{J}$.

We utilize the MPC based approach stated in Algorithm 3. In this approach, a dynamical system model is used to predict the system behavior with respect to time so as to generate an optimal sequence of control inputs for a given time horizon (M). More specifically, we use this method to generate a sequence of control (button pressed or not pressed) that minimize the objective function, that is, to minimize the amount of push buttons. At a given time $t \in [0, T]$, we solve the optimization problem in (10) over a given prediction horizon, N, and apply the obtained input sequence for the amount of the control horizon M. In this optimization scheme, it is necessary to note that the prediction at t_{k+1} depends on the current state variable and the applied input at time t_k. Moreover, to guarantee feasibility, one has to tune M, N, and the sampling time. For example, a small value of sampling time will result in a more transient response with cost of computation time. Similarly for a fixed N, a smaller M result in a conservative input that maximize the safety criteria, however slower output response since prediction has to be done more frequently.

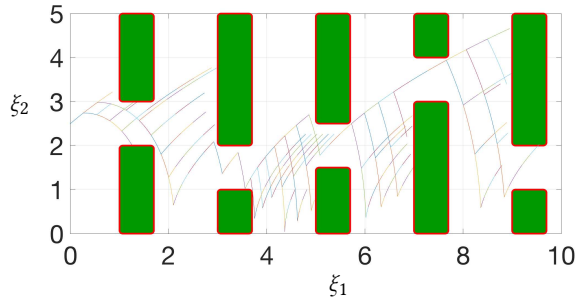

Figure 9: Nonuniqness of solutions for Flappy Bird game variants with different jump time, $j(t)$, and total number of jumps J. The result shown in this simulation uses a receding horizon of 10 time steps and control horizon of 7.

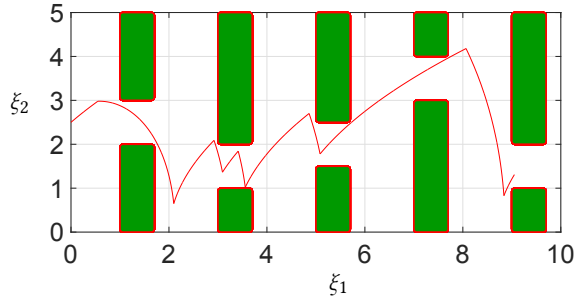

Figure 10: Optimal trajectory that minimizes the number of jumps required to travel through the game space while abiding by the system constraints.

Numerical experiments were carried out to analyze solutions for the developed strategy of motion planning.[5] In order to guarantee the existence of solution for MPC problem, a careful initial condition selection is necessary. For example, for the hybrid model of (4), an initial condition pertaining to the variable q could lead to no solution. That is, the initial condition could trap the character outside reachsafe$^{\mathcal{H}}_{T,J}(\chi_o)$. Therefore, we utilize Algorithm 3 in Section 4.2 to compute the appropriate initial condition (for part of the state variable) that results in optimal solution. Moreover, we utilize the results obtained in Algorithm 1, to speed up the MPC by constraining the search only inside reachsafe$^{\mathcal{H}}_{T,J}(\chi_o)$.

In Figure 9 , trajectories satisfying dynamical and terminal constraints listed in (10) are depicted. That is, trajectories reach the target point $\xi_1 = 9.7$ and $\xi_2 \in [0, 5]$ with minimal state transition are shown.

6 CONCLUSION

In this article, we proposed a framework for game design and quantitative analysis in order to guarantee the best compromise between the difficulty of the game, the fun and the excitement acquired. After noticing that the complexity of the game character's dynamics as well as the game environment make the design process increasingly complicated and time consuming, as opposed to the traditional iterative processes to manually tune the game model parameters, we proposed a mathematical framework that interpret (model) the game evolution with respect to time in terms of mathematical equations (differential and difference inclusions). The resulting models belong to the general class of hybrid dynamical systems. We then formulated the game objectives in terms of the solutions to the obtained hybrid system model. More specifically, we used the reachability, feasibility and optimality concepts to guarantee the playability of the game as well as to quantify the difficulty of the game level.

One way to compute the reachable set when inputs of the system are discrete and take finite countable values is to simulate the system for all possible input sequences. However, computing all the trajectories from a given initial condition can be a very time consuming task. Even in the case, where the inputs are boolean (1 or 0), as in the case of Flappy Bird, the number of input sequences is exponential of the form 2^N, where N is the size of the input sequences. To address this inadequacy, some algorithms are proposed in the literature.

Our approach is generic and covers different types of action/arcade style games with a different level of complexity. The proposed approach can handle games with dynamic environments such as different levels of Mario or complex version of Flappy Bird where traditional path planning can fare poorly. Furthermore, an automatic design process preserving the specified difficulty level can be achieved using the proposed tools.

Future work includes testing the effectiveness of the proposed study on various games in both 2d and 3d while analyzing the computational burden, investigation on more elaborated and efficient approaches to estimate reachable sets with less computational burden, and proposing (and evaluating) new metrics that accurately evaluate the difficulty of the game with respect to its different parameters (obstacles, character speed, etc).

[5] Source code available at https://github.com/HybridSystemsLab/FlappyBirdPlanning

REFERENCES

[1] Berk Altın, Pegah Ojaghi, and Ricardo G Sanfelice. 2018. A Model Predictive Control Framework for Hybrid Dynamical Systems. *IFAC-PapersOnLine* 51, 20 (2018), 128–133.

[2] B. Altin and R. G. Sanfelice. 2019. On Model Predictive Control for Hybrid Dynamical Systems. In *To appear in American Control Conference*.

[3] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*. Springer, 209–229.

[4] J-P Aubin, John Lygeros, Marc Quincampoix, Shankar Sastry, and Nicolas Seube. 2002. Impulse differential inclusions: A viability approach to hybrid systems. *IEEE Trans. Automat. Control* 47, 1 (2002), 2–20.

[5] Aaron William Bauer and Zoran Popovic. 2012. RRT-Based Game Level Analysis, Visualization, and Visual Refinement.. In *AIIDE*.

[6] Alberto Bemporad and Manfred Morari. 1999. Verification of hybrid systems via mathematical programming. In *International Workshop on Hybrid Systems: Computation and Control*. Springer, 31–45.

[7] R. Goebel, R. G. Sanfelice, and A.R. Teel. 2009. Hybrid dynamical systems. *IEEE Control Systems Magazine* 29, 2 (April 2009), 28–93. https://doi.org/stamp/stamp.jsp?tp=&arnumber=4806347&isnumber=4806311

[8] Rafal Goebel, Ricardo G Sanfelice, and Andrew R Teel. 2012. *Hybrid Dynamical Systems: modeling, stability, and robustness*. Princeton University Press.

[9] Aaron Isaksen, Daniel Gopstein, and Andrew Nealen. 2015. Exploring Game Space Using Survival Analysis.. In *FDG*.

[10] Sertac Karaman and Emilio Frazzoli. 2010. Optimal kinodynamic motion planning using incremental sampling-based methods. In *Decision and Control (CDC), 2010 49th IEEE Conference on*. IEEE, 7681–7687.

[11] John Lygeros, Karl Henrik Johansson, Slobodan N Simic, Jun Zhang, and Shankar S Sastry. 2003. Dynamical properties of hybrid automata. *IEEE Transactions on automatic control* 48, 1 (2003), 2–17.

[12] Tim Mercy, Wannes Van Loock, and Goele Pipeleers. 2016. Real-time motion planning in the presence of moving obstacles. In *Control Conference (ECC), 2016 European*. IEEE, 1586–1591.

[13] Anthony N Michel and Bo Hu. 1999. Towards a stability theory of general hybrid dynamical systems. *Automatica* 35, 3 (1999), 371–384.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[15] J. C. Osborn, B. Lambrigger, and M. Mateas. 2017. HyPED: Modeling and Analyzing Action Games as Hybrid Systems. In *Artificial Intelligence and Interactive Digital Entertainment Conference*.

[16] Ricardo Sanfelice, David Copp, and Pablo Nanez. 2013. A toolbox for simulation of hybrid systems in Matlab/Simulink: Hybrid Equations (HyEQ) Toolbox. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*. ACM, 101–106.

[17] Ricardo G Sanfelice, Michael J Messina, S Emre Tuna, and Andrew R Teel. 2006. Robust hybrid controllers for continuous-time systems with applications to obstacle avoidance and regulation to disconnected set of points. In *American Control Conference, 2006*. IEEE, 6–pp.

[18] Noor Shaker, Mohammad Shaker, and Julian Togelius. 2013. Ropossum: An Authoring Tool for Designing, Optimizing and Solving Cut the Rope Levels.. In *AIIDE*.

[19] Adam M Smith. 2013. Open problem: Reusable gameplay trace samplers. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[20] Adam M Smith, Mark J Nelson, and Michael Mateas. 2009. Computational Support for Play Testing Game Sketches.. In *AIIDE*.

[21] Gillian Smith, Mee Cha, and Jim Whitehead. 2008. A framework for analysis of 2D platformer levels. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*. ACM, 75–80.

[22] Steve Swink. 2009. *Game feel: a game designer's guide to virtual sensation*. Morgan Kaufmann.

[23] L Tavermini. 1987. Differential automata and their discrete simulations. *Non-Linear Analysis* 11, 6 (1987), 665–683.

[24] Claire J Tomlin, Ian Mitchell, Alexandre M Bayen, and Meeko Oishi. 2003. Computational techniques for the verification of hybrid systems. *Proc. IEEE* 91, 7 (2003), 986–1001.

[25] Arjan J Van Der Schaft and Johannes Maria Schumacher. 2000. *An introduction to hybrid dynamical systems*. Vol. 251. Springer London.