Softlock Detection for Super Metroid with Computation Tree Logic

Ross Mawhorter
rmawhort@ucsc.edu
University of California, Santa Cruz
Santa Cruz, CA, USA

Adam Smith
amsmith@ucsc.edu
University of California, Santa Cruz
Santa Cruz, CA, USA

ABSTRACT

Videogame level designs can contain errors called *softlocks* where a player traversing the level in an unintended manner can become permanently stuck. In this paper, we explore the automated detection of softlocks in the game *Super Metroid* using Computation Tree Logic (CTL). *Super Metroid* distinguishes itself as an example domain because of its velocity-based movement and rich item upgrade hierarchy. These factors can cause softlocks in Super Metroid to be challenging to detect visually. We contribute a tile-based gameplay abstraction for *Super Metroid*, and demonstrate verification of CTL properties for scenarios based on a segment of the original game's level design. CTL can be used to define and test many other gameplay properties (e.g. which bosses can be skipped or which order items may be collected) and is immediately applicable to other game designs for which a compact abstraction of their state space can be enumerated. By making plausible design changes to a *Super Metroid* level fragment, we show how highly nonobvious softlocks can be detected and how the counterexamples resulting from verification failure can be turned into visualizations that explain the problem.

KEYWORDS

metroidvania, machine playtesting, state abstraction, model checking, formal verification

ACM Reference Format:
Ross Mawhorter and Adam Smith. 2021. Softlock Detection for Super Metroid with Computation Tree Logic. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021 (FDG'21), August 3–6, 2021, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472538.3472542

1 INTRODUCTION

Imagine for a moment you are playing through a set of machine-generated *Super Mario Bros* [11] levels, and you come across the Mario level shown in Figure 1. If you make the jump, you can complete the level by reaching the flag. However, if you fail the jump, you will be "softlocked," or permanently stuck. That is, you are trapped in the gap with no way to make forward progress or even restart the level by touching an enemy or falling into a pit. Your only option is to reset your console and potentially lose a significant amount of progress. If you get stuck multiple times, you might become frustrated and give up altogether. It might also be challenging to know whether you are softlocked because softlocks can be much harder to spot than a pair of walls that are too high to jump. This paper introduces the use of Computation Tree Logic (CTL) to state a formal definition of softlocks and to find potential softlocks in the level designs of a non-*Mario* game: *Super Metroid*.

This paper focuses on detecting softlock conditions that might arise when making changes to level designs for the one specific commercial videogame *Super Metroid* [14]. Although the game was originally released in 1994, fan-created modifications[1] (ranging from minor improvements to art or game mechanics to total conversions) are being continually produced decades after the commercial release. However, this focus is also part of a larger effort to improve the depth of technical discourse in academic game AI by moving away from *Mario* as the default motivating example. In the same way that Chess was once considered "the Drosophila of AI" [15] (a model organism to standardize experimental methodologies), variations and clones of 1985's *Super Mario Bros* have anchored many academic game AI projects [10] including game playing [9], level generation [8], player modeling [18], design assistance tools [7], game space visualization [1], and game moment retrieval [28]. Our shift in attention from *Mario* to *Metroid* (while still admittedly offering one aging videogame to stand for in all others) is intended to break to two assumptions that prevent *Mario*-based work from being cross-applied to other games.

Space is time: In *Mario*-style games (including the vertically-oriented game *Kid Icarus* [12]), progress over time in completing a game level strongly correlates with movement through one direction in space. That Mario's horizontal position is such a good proxy for progress leads to an algorithm as simple as A* achieving competition-winning gameplay performance [24], even exhibiting super-human finesse (e.g. in wall-jumping). Although the player is technically able backtrack a bit in *Mario* levels, it is rarely ever required. In *Metroid*-style games, by contrast, the player is asked to find their way through a world that requires traversing and backtracking in all spatial directions. Our focus on *Metroid* decouples the roles of time and space while still retaining a focus on level designs encoded as a two-dimensional grid of tiles (which has been useful for making connections to the fields of computer vision [26] or natural language processing [8]).

Tile space is state space: Speed and direction of movement are important concepts in many platformer games. However, progress through a *Mario* level can be modeled well enough without these concepts, so we often think of the game as having a two-dimensional

[1]https://metroidconstruction.com/hacks.php

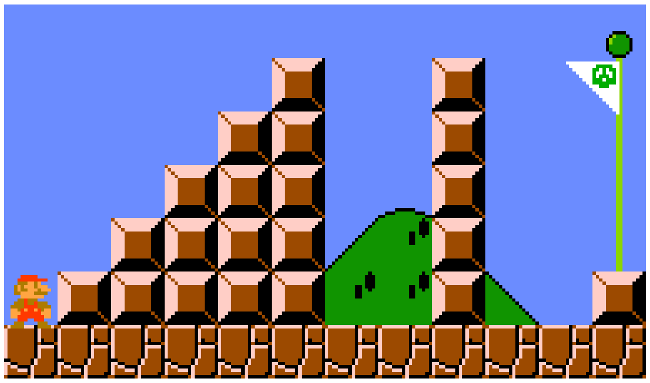

Figure 1: A hand-constructed example Mario level that contains a *softlock*: when Mario falls into the gap between blocks in the middle of the scene, he can no longer jump out. Since there are no enemies or pits, the player has to manually reset the console to escape.

state space. Some gameplay validators for procedurally-generated levels, for example, judge a level as playable if a path can be drawn on the level's two-dimensional tile grid connecting the start to the finish tile [4]. In *Metroid* games (or faster paced platformers like 1991's *Sonic the Hedgehog* [5]), velocity plays a much bigger role in defining which paths through a level are feasible. *Metroid* games (and other games like 1986's *The Legend of Zelda* [13]) involve collecting items and abilities over time so that the player is encouraged to re-traverse the same spaces repeatedly as they find uses for their new abilities. Adding velocity moves *Metroid* gameplay into a four-dimensional state space and consideration for different combinations of collectable items expands this manyfold. While the *mostly* two-dimensional nature of *Mario* games has made for convenient rendering of figures in papers, a focus on *Metroid* forces us to directly confront this higher-dimensional, abstract state space. Problems with a level design, such as a softlock, might be associated with particular locations in the game's two-dimensional tile grid, but they are not detectable without careful analysis of non-local properties of the broader abstract state space.

The core contribution of this paper is a way of formalizing design issues (like the existence of softlocks) as CTL properties that will be evaluated over the game's entire abstract state space. We also contribute a tile-based abstraction of *Super Metroid* to parallel the tile-based movement modeling often applied to *Mario* games. The inclusion of velocity and collected items in our abstraction of *Metroid* allow CTL statements to express complex, non-local properties of game levels.

2 BACKGROUND

This paper builds on prior work applying formal methods to games research. This section outlines some of the related contributions in this area, then discusses the mathematical formalisms used for verifying *Super Metroid* level designs.

2.1 Prior Work

In this paper we are concerned with verifying properties about possible gameplay traces in a single-player game. In the technical games research literature, Lee et. al. recently considered the problem of defining a platformer game movement model in order to check reachability, but do not attempt to verify more complex properties [25]. To do this, they describe a fine-grained game abstraction which has both position and velocity as well as other discrete state (such as whether an optional objective has been reached). There has been other work that focuses primarily on the abstraction of games to formal spaces without concern for verification. Cook et al. developed a way to reduce the size of the abstracted state space without sacrificing fidelity, but they focus on games with much smaller state space than most platformers [3]. Osborn et al. offered a (hybrid, discrete-continuous) formal model that considers variables like position and velocity as continuous variables governed by differential equations [17]. This technique could be used to solve reachability problems at high precision, but it limited applicability of analyses that assume finite state spaces.

The above mentioned work considers abstracted models of existing games where there might be a gap between what is possible in a model versus the game that inspired the model. In contrast, the LUDOCORE system represented games as logical systems [22], so that human gameplay corresponded exactly with the formal system used for analysis. In exchange for closing the modeling gap, this approach limited the range of expressible games to those with mechanics specified using logical predicates.

Outside the world of verification, Silva et al. use AI-based playtesting, and find problematic gameplay scenarios in a board game using tens of thousands of simulated games [6]. This type of simulation holds promise for platformer videogames as well, but there are several challenges. Primarily, developing an agent that can play the game competently is much more difficult because it needs to interact with the world frame-by-frame. Additionally, relying solely on simulation data to fix bugs in level design is problematic. For example, even if the AI agent never reaches a certain section of the level, that does not necessarily imply it is impossible (or even difficult) for a human player to go there.

The systems we have examined so far have mostly been concerned with showing that there exists at least one key gameplay trace (such as one reaching from start to finish) or ensuring no single problematic gameplay trace exists (such as one that reaches the finish in fewer steps than expected). The strategy of logically *quantifying over play* suggested in the analysis (and synthesis) of *Refraction* puzzle levels [21] foreshadows the mode of analysis presented in this paper. Critical puzzle design properties can involve logical quantifiers beyond a single $\exists$ or $\forall$ quantifier. Since generated puzzles can be solved in many ways, the authors used formal logic to ensure that all of the solutions to generated puzzles required practicing a key concept that the game wants to teach (synthesizing puzzles to satisfy an $\exists\forall$ specification). Although this work considered all possible puzzles and piece configurations, it did not consider how the player would come to reach a given configuration by laying one piece down at a time. In our analysis of *Metroid* levels, the steps on the path leading to a critical state (e.g. one manifesting a softlock) can provide important information about how to change to the level to eliminate it.

2.2 Kripke Structures and CTL

In this paper, we model gameplay as a Kripke structure[2, a.]:

$$M = (S, I, R, L)$$

Here, S denotes the set of all possible states, I denotes the set of initial states and $R \subseteq S \times S$ is the transition relation. R defines for any given a state s_i, which states s_j can be reached directly from s_1. The labeling function $L : S \to 2^{AP}$ uses a finite set of atomic propositions (AP) to provide semantic information about each state. This is essentially a graph of labeled vertices over the underlying state space of the game.

Computation Tree Logic (CTL) [2, b.] is a branching-time logic that models properties applying to possible paths within M by using the set of atomic propositions AP. CTL has two key modal operators (along with the usual Boolean connectives $\neg \wedge \vee$): AG (sometimes written as $\Box$) which means "for all states globally" and EF (sometimes written as $\Diamond$) which means "there exists a path such that finally". The property $AG(p)$ holds at a state s if and only if p holds at every point on every path that starts at s. The property $EF(p)$ holds at a state s if and only if there exists some path starting at s such that p eventually holds on that path. A property is said to hold for a system M if and only if it holds at every initial state $s \in I$. AG and EF are not the only modal operators in CTL, but they are sufficient for examination of softlocks and related phenomena.

Since we are primarily interested in softlock detection, we can encode this problem in CTL as follows. First, we add a label *goal* to the set of atomic propositions and apply it to the set of ending states for a level. This set of ending states can be defined by a certain position, but can also include various other factors such as having defeated a certain boss. Now we formulate the "no softlocks" property as:

$$P = AG(EF(goal))$$

In plain English, this is saying it is possible to reach the goal state from every reachable state. Working from the inside out, *goal* says the current state is a goal. $EF(goal)$ says it is possible to reach a goal state from the current state. $AG(EF(goal))$ says that from all states reachable from the current state, a goal state is possible to reach. If P holds for all initial states in M, we can say that the game that M represents is free of softlocks.

To show how it applies, consider the example shown in Figure 1. We label all states where Mario is touching the flag with *goal*. At the bottom of the pit, the property $EF(goal)$ does not hold because there is no path to the flag. The property $AG(EF(goal))$ does not hold at the (singular) starting state because it is possible to reach the bottom of the pit where $EF(goal)$ does not hold. Thus, P does not hold for the structure M. From a formal verification perspective, this system has simply failed verification. From a human design assistance perspective, however, this is an opportunity to do additional reasoning to highlight for a designer where this softlock can occur and which style of play could give rise to it.

This formulation is not the only possible definition for softlocks. If a game doesn't have a clear goal, it is also possible to define a $start$ label in a similar way and enforce $AG(EF(start))$ to ensure that the player will not get stuck in a part of the level separated from the starting point.[2] In this paper we focus on $AG(EF(goal))$ as an important special case of applying CTL formulas to games. CTL is flexible, and it can also express many other desirable properties.[3]

3 LEVEL DESIGN FORMALISM

In this section we discuss a tile-based abstraction of *Super Metroid* which can be used for softlock detection. This abstraction is hand-authored (i.e. not automatically derived from the game's executable format or source code), and the techniques used to create it are also discussed. The code for this project is available at https://github.com/aremath/sm_rando/blob/master/world_rando/model_checking.py.

3.1 Motivation for Abstraction

Abstraction is a commonly-used verification technique that reduces the problem size. For verifying systems formally using CTL, the usual requirement is that the abstracted system *bisimulates* the original system [2, c.]. This means that the original system and the abstraction will satisfy the same CTL properties. The abstraction used in this paper does not attempt to bisimulate *Super Metroid*, and so the abstract version of a level may not satisfy all properties of the original level in the base game. It is thus important to motivate the reasons behind using such an abstraction, and discuss how it can still be useful despite these shortcomings.

The primary motivation behind constructing this abstraction is feasibility. In order to verify properties about *Super Metroid* levels without abstraction, we would need to verify properties of the reachable states at the level of the game console hardware state (e.g. hundreds of thousands of bytes of main memory, various system clocks, and sundry subsystems such as the audio processing unit). High-fidelity hardware-modeling emulators for the Super Nintendo Entertainment System (SNES) exist (e.g. higan[4]), and *local* state space exploration using this level of simulation has been used in previous work [1, 27]. The CTL formalism, however, is most useful when the *entire* state space can be explored. (The SNES has 128 KiB of RAM, making the possible state space preposterously large.) Even considering just the reachable states, trying to represent this space in terms of ground truth hardware is completely impractical. Typical abstraction techniques require building the system before creating the abstraction, so even creating a bisimulative abstraction of the true game state is infeasible.

Meanwhile, even if it were possible to represent the entire transition system of the game, such a system would not be ideal for verifying properties of videogames played by human players. For example, by simulating a system that allows frame-by-frame input,

[2]The $start$ label would probably be defined in such a way as to include all game states where the player is in the starting location, not just in one specific starting state. It could be enough that the player can return home, not requiring them to also shed any special statuses they collected when they were away.

[3]For example, it is possible to verify that a certain item must be obtained in order to pass a certain door. To do this, introduce a label $door$ and apply it to all states that have positions immediately past the door. Then introduce a label $item$ which applies to all states where the player has that item. Now we can verify the property by checking whether the level satisfies $AG(door \to item)$.

[4]https://higan.dev/

the system could verify reachability properties that require frame-perfect timing. Concretely, a level might pass softlock verification only to manifest a softer sense of softlocking where no human hand is capable of executing the required (and technically feasible) moves to get unstuck. Hard-to-execute and counterintuitive glitches in the game would also allow the system to verify reachability properties using paths that the average player of the game would never even imagine trying to accomplish. Finding these bugs in the game is a worthy aim, but it is not necessary in order to detect situations as in Figure 1.

That a level design is free from softlocks is a fact about the player experience within the level. Thus we want to verify the set of *playable* traces, and we use abstraction to define that set. This abstraction will not model the original game exactly, but it will still be able to provide useful feedback to level designers. The abstraction created for this paper was designed by hand; the future work section discusses the possibility of using actual play data in order to define the abstraction automatically.

3.2 Abstraction

We begin by drawing a distinction between the *fundamental rules* of the game (which are fixed), the particular *problem instance* under consideration (which varies across applications of the algorithm), and the *player state* within that instance, (which varies within a single application of the algorithm). Separating the fundamental rules from the problem instance is important because multiple level designs for the same game should be analyzed under the same rules. For platformers, the player character's position and velocity are part of the player state, the level design that the player moves within is a problem instance, and the collision detection rules are part of the fundamental rules. In a *Zelda*-like game, the number of keys obtained and the set of opened doors would be player state variables since they will alter during a single playthrough of a problem instance. The placement of the walls and doors is part of the problem instance because it differs between level designs but is static within a playthrough. Finally, the rules that prevent the player from walking through walls are part of the fundamental rules, which are the same for each level design.

Each state in the Kripke structure corresponds to a player state from the game. The problem instance and the fundamental rules together govern which transitions are possible. For example, if walls cannot be changed by the player (part of the instance rather than the player state), the set of walls is used to disallow transitions between states where the player would travel through a wall. In this way, the level geometry and other aspects of the problem instance are implicitly encoded in the transition relation. While a complete model would include enemies and other moving entities we ignore them here for the sake of simplicity.

Like many platformers, *Super Metroid* has tile-based level geometry, so we can abstract the player position to a single tile rather than a pixel or subpixel position. We can also abstract the player velocity to tiles as well. After running for three tiles, the player has some internal horizontal velocity. In the abstract, we can denote this as velocity three. Running right (or left) one tile increases (or decreases) the player velocity by one, up to a maximum determined by the in-game maximum running speed. This abstraction ensures that movements will be consistently possible while also dramatically reducing the set of possible velocities. Since *Super Metroid* has multiple physics schemes (for example, if the player is underwater), we add a discrete variable to the velocity representing what *kind* of velocity it is. Two tiles of underwater velocity is treated as completely distinct from two tiles of velocity gained by running when the player is in the air. Player velocity and position are the only two numerical state variables (each represented in-game as 4 bytes per coordinate, or 8 bytes total). The other player state variables can be modeled discretely without abstraction. In our model these are the player pose and the item set.

Pose determines the player hitbox, and whether the player is currently jumping. In *Super Metroid*, the player cannot jump again in mid-air without the Space Jump powerup, so the player state must keep track of that. The player hitbox can also change from a 1×3 tile standing pose to a 1×2 spin-jumping pose, and a 1×1 ball pose. This allows the player to access different sizes of passageway and interact with the level geometry in different ways.

Items are the primary way that players gain abilities to interact with the game world. Once an item has been picked up, a player has the corresponding ability forever. These can range from the ability to destroy new block types, to the ability to jump multiple times in midair.

The set of possible player states S is thus the Cartesian product of the (discretized) position and velocity with the player pose and the player item sets. Since CTL properties hold over paths from an initial state, we can restrict S to the set of states reachable from the initial states I without changing which properties will be satisfied.

3.3 Representing Abstract Movement

To formulate a Kripke structure, we must now build R, the transition relation. This means constructing a model of how the player moves through the level. Whether a player can perform an action depends on both the player state and the problem instance. For example, using a key to open a door requires both that the player has an appropriate key, and that there is a door adjacent to the player. A movement rule can be thought of as a function which maps a set of input configurations (including both player state and problem instance) to a set out output states, which are new states the player can transition to if the conditions are met.

Because the fundamental movement rules for *Super Metroid* do not depend on the player position, we can formulate them without reference to a specific problem instance. Instead, they depend on only local features of the problem instance (i.e. which tiles are solid or not near the player) to determine whether an action is possible.

This reduces the problem of creating R to one of defining these movement rules in terms of their domains and ranges. While in theory these can be arbitrary sets of states, they will typically fall into simpler categories (such as rules that can only be applied when the vertical velocity is ≥ 0). To this end, we created a Domain-Specific Language for specifying rule domains and ranges. In general, the outcome of a rule can depend on different geometries in the domain. For example, applying a single jump rule can have different results depending on whether the player would collide with a wall during the jump.

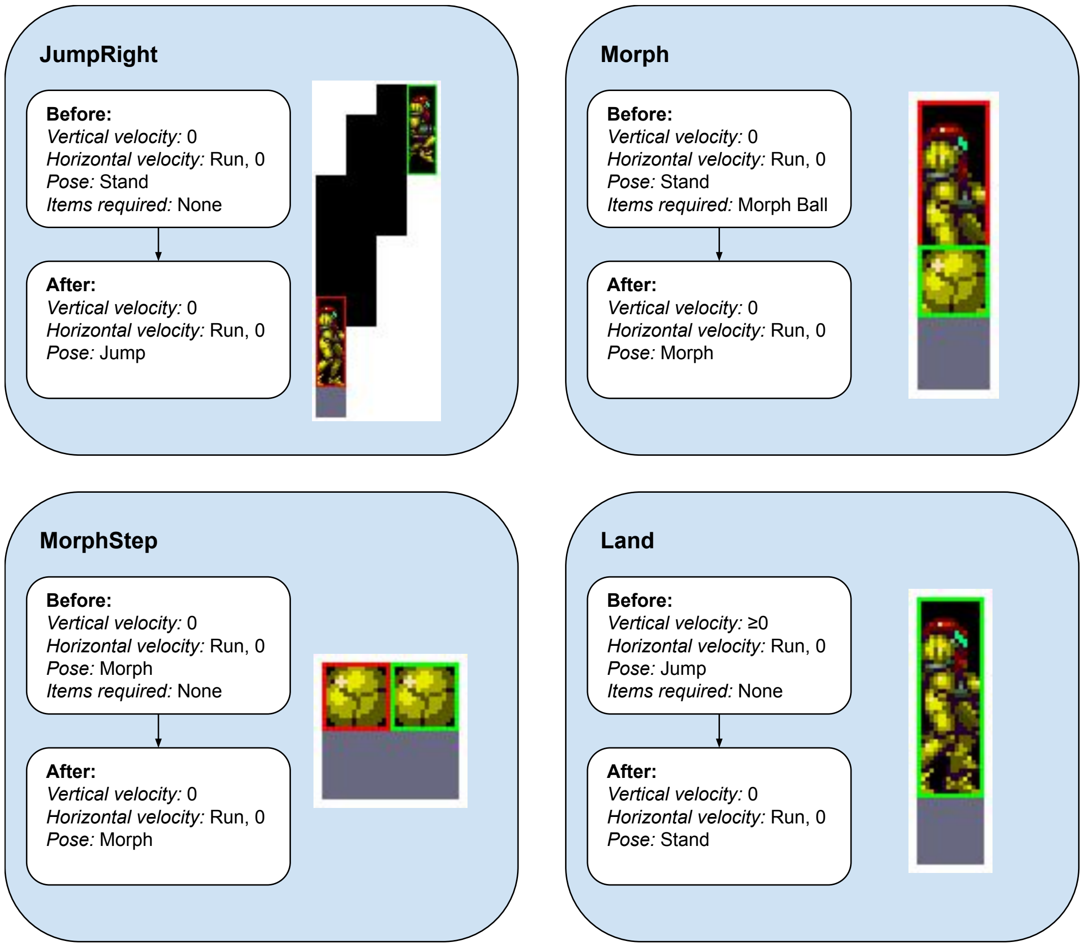

Figure 2: Four example movement rules which are parsed by the system. Each rule consists of a text description of the non-spatial domain and range along with an image that describes the spatial aspect of the domain and range. For the image, gray represents solid tiles, black represents air tiles, and white tiles are unspecified (can be any tile). The starting player position is outlined in red, and the ending position in green. The gray and black tiles specify the local problem instance domain.

Figure 2 shows four example transition rules. Each rule is defined using both an image and text. The image captures the local level geometry and the relationship between the initial and final player positions. The text captures the non-spatial aspects of the rule: the initial and final possible player velocities, poses, and item sets.[5] In these examples, the player velocity is mostly unchanged: for example, in **MorphStep**, the player can achieve maximum rolling speed across a single tile, so the player velocity is constant even though this rule represents horizontal movement. In the **Land** example, the player position does not change. This rule specifies how a falling player can lose all of their vertical velocity to land on the ground, as long as their initial velocity is positive (towards the ground). This image-based language makes it easy to author movement rules. We also used this language to encode level geometries and their start/goal states.

[5]For more information, refer to https://github.com/aremath/sm_rando/blob/master/encoding/rules/rules.yaml

After parsing each of the given transition rules, the Kripke structure can be built. First, we search over application of the movement rules to find the set of reachable states. In the Kripke structure $M = (S, I, R, L)$ this set of reachable states is S. Then, for each reachable state, we attempt to use every possible transition and record the successful transitions as R. We use the user-provided definition of I as the starting state, and we label all user-specified *goal* states, which defines L. This process will also implicitly encode the level geometry, since rule application will fail when the local level geometry is not in the rule's domain.

4 VERIFYING LEVEL DESIGNS

Once the Kripke structure for a given set of movement rules and given level geometry has been obtained, we can use an off-the-shelf model-checker to determine whether the level satisfies properties in many logics. This section uses a detailed example to show how this verification works and how the system finds and explains counterexamples when the desired property is not satisfied.

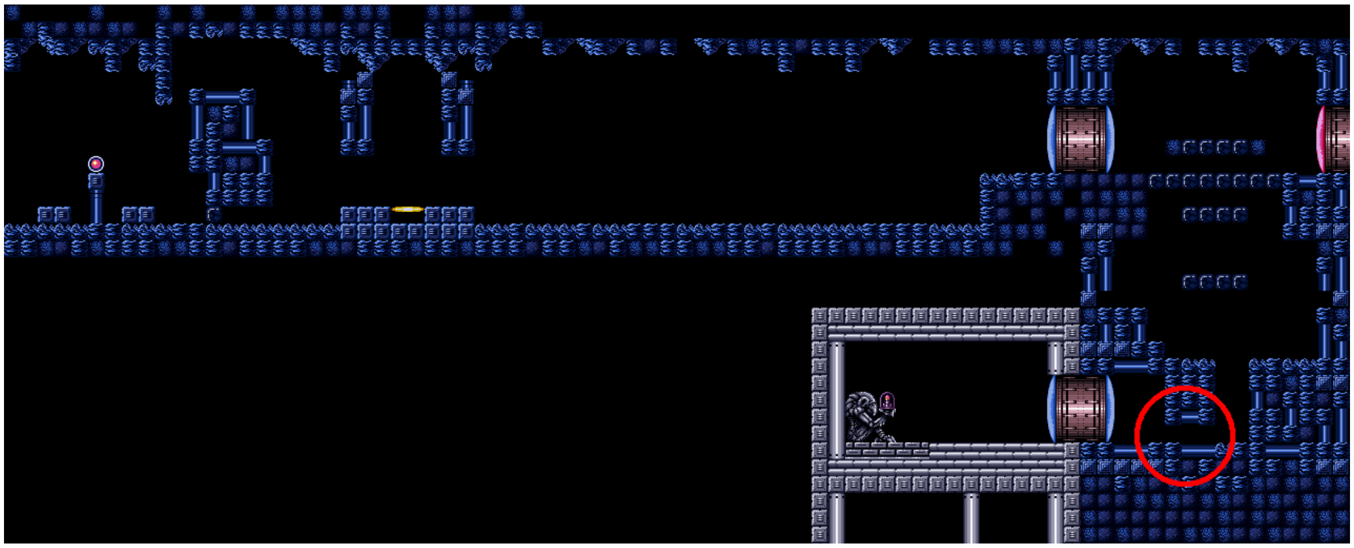

Figure 3: A fragment of level design from *Super Metroid*. For the purposes of this paper, we model the initial state as standing on the bottom of the elevator (the yellow pad) with no items. The ending state is entering the door transition at the upper right (past the pink missile door). To enter the ending state, the player must acquire the morph ball (left). They then travel to the right and down to obtain the missiles (bottom), then climb back up and destroy the door cap with the missiles. The red circle indicates a tight passageway where the morph ball must be used before obtaining the missiles.

4.1 Verification Example

To demonstrate how the system works, we focus on a single portion[6] of the *Super Metroid* map shown in Figure 3. In this level, there are two items, the morph ball at the top left, and the missiles at the bottom right. The morph ball allows the player to enter small passageways, and the red circle in Figure 3 shows where the player must use it (squeezing through a narrow passage) to obtain the missiles. This missile upgrade allows the player to destroy the missile door (pink) in the top right and reach the *goal*. Because the player state includes the items the player has collected, the shortest path to reach the *goal* state actually starts out by traveling left to obtain the morph ball. In this example, there is a single *goal* state, but in general many *goal* states can be specified.

As in the original game, it is possible to get from the *start* state to the *goal* state. In the language of CTL, this design satisfies $EF(goal)$. In the terms used by past work on Mario level generation, this level is *playable* (or, *winnable*).

Beyond this basic fact, there are many useful properties we can verify about this level design. For example, it also has the property of $AG(goal \longrightarrow morph)$. That is, entering the *goal* state implies that the player has obtained the morph ball. Finally, we can verify the "no-softlocks" property: this level actually does satisfy $AG(EF(goal))$ in the abstraction described in section 3. Not only can the player reach the goal, it is impossible for them to reach a state from which the goal is unreachable. By contrast, the level design sketched in Figure 1 would be considered playable despite the potential for softlocking.

[6]This portion represents the areas known to the speedrunning community as the *Morph Ball Room*, *Construction Zone*, and *First Missile Room*. See https://wiki.supermetroid.run/Morph_Ball_Room

To see how softlocks can be detected, we introduce a plausible softlock to this level as shown in Figure 4. Figure 4(a) shows one modification to the original level (allowing the player to explore a new passageway in the bottom-left while in the morph ball pose). By editing just six more tiles (opening up the tight passageway near the missile powerup), we arrive at Figure 4(b). Neither local change introduces a softlock by itself, but the combined changes do. Surprisingly, the softlock manifests in the region of the first change only after a seemingly unrelated and small change is made elsewhere. As shown in Figure 5, the player can now find themselves in the new narrow passageway on the far left having picked up missiles but not the morph ball. Unable to jump up or roll out (in ball form), they are softlocked. While the second change might be seen as forgiving or permissive (it allows the player more freedom to collect powerups in the order they choose), it actually permits them to get themselves in trouble without hope for recovery. While it is no surprise that small level design changes can have large impacts on the player experience, this example demonstrates the elusiveness of potential softlocks. Even in a nominally two-dimensional platformer game, the combinatorial nature of backtracking through locations with various item combinations can exhaust a designer's ability to identify potential softlocks.

4.2 Visualizing Counterexample Traces

When a model does not meet a verification standard, the usual course of action is to provide a counterexample that demonstrates the failure. For many CTL properties, it is not obvious how to show a counterexample that disproves the property. For example, if we have the property $EF(goal)$, a disproof would require showing that

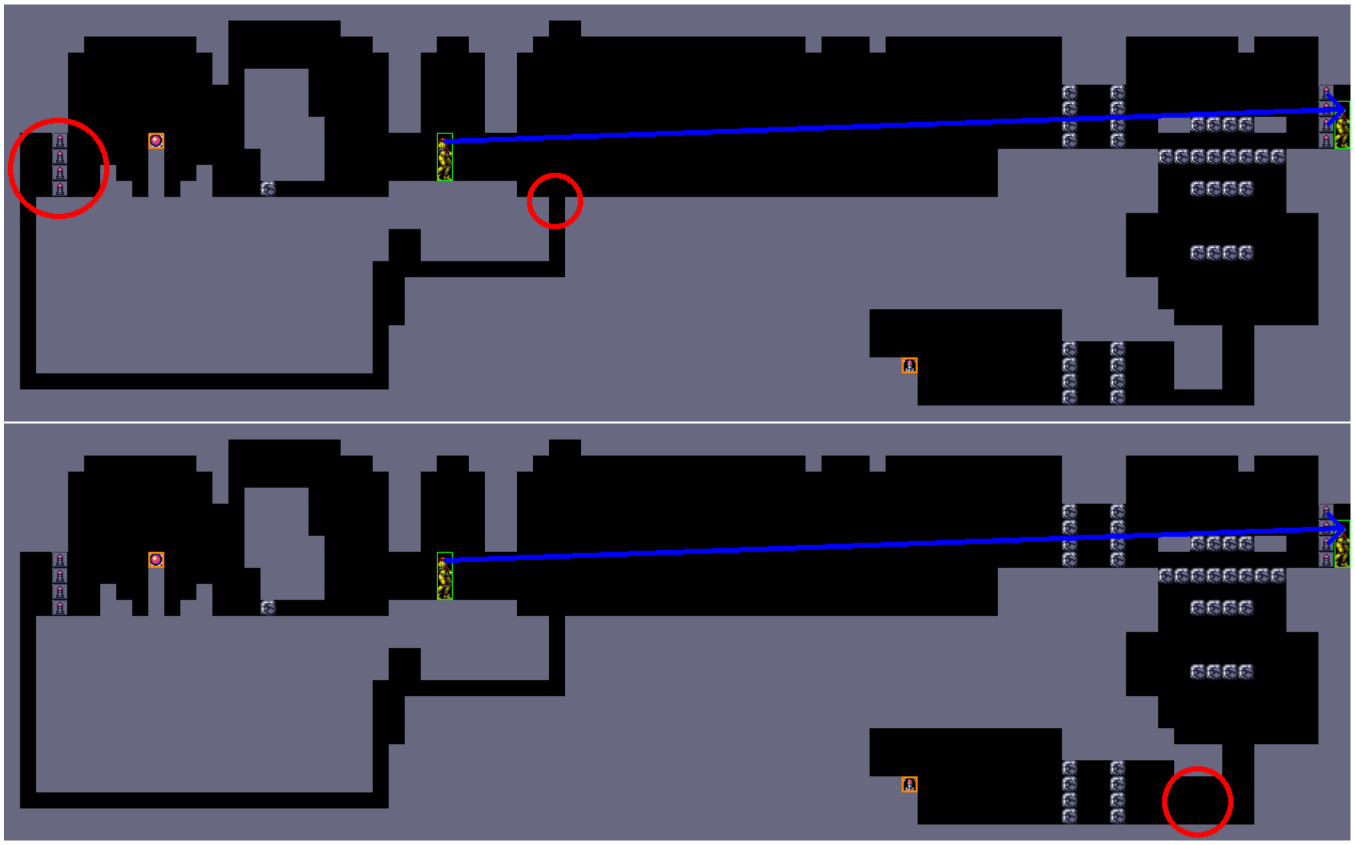
Figure 4: Two modified abstract levels (a) and (b) based on the real level shown in Figure 3. The path on the left (entrances circled) has been added to illustrate how the system detects softlocks. In each level, the blue arrow points from the *start* state to the *goal* state. Design (b) is identical to (a) except where circled. Design (a) satisfies $AG(EF(goal))$, while design (b) does not. Figure 5 shows why (b) fails this verification. The two item upgrades (missiles and morph ball) are highlighted in orange.

$AG(\neg goal)$ (all reachable states do not have the *goal* label), which does not have a simple visual interpretation.

However, for the general property $AG(p)$, a counterexample consists of a sequence of gameplay actions that terminates in a state where $EF(p)$ does not hold. For softlock detection, this means finding a path to a state where the goal is no longer reachable. By studying the path, a level designer can learn why the property was violated, and understand what changes they should make to the level in order to fix it. Figure 5 is a trace that shows why Figure 4(b) does not satisfy $AG(EF(goal))$. In this trace, the player accesses the missiles before getting the morph ball. Because of the higher ceiling created by the the edit circled in red in Figure 4(b), the player can simply walk through to acquire the missiles. After that, they can jump over the morph ball and enter the passageway that requires missiles to enter but morph ball to escape, becoming stuck.

In this example, a designer can see that the root cause of the softlock was enabling access to missiles before morph ball. While they can fix this by editing the passageway back to the way it was in Figure 4(a), any other change they make that ensures the player will have morph ball before entering the lefthand passageway would also work.

4.3 Computational Efficiency

The efficiency of this system depends on many factors. The size of the Kripke structure that is used to model gameplay depends both on the size of the level and the number of transition rules in the player movement abstraction, as well as other factors like the number of item upgrades present. Once the structure is built, the desired CTL property must be verified. This depends on the size and complexity of the property, as well as the speed of the software used to verify the Kripke structure. Formally, the algorithm for checking whether a given Kripke structure $M = (S, I, R, L)$ satisfies a given CTL formula p has time complexity $O(|p| \cdot (|S| + |R|))$. For this system, however, the number of states can be quite large. Since our abstraction models player velocity and item sets, the number of states per position could depend exponentially on these other factors.

Despite these potential problems, the system can verify our example levels quite quickly in practice. Table 1 summarizes the running time information for verifying these examples. All tests were performed on a personal laptop with an Intel i7 processor and 16 GB of RAM using an unoptimized single threaded Python program for search. After enumerating the state space and its transitions,

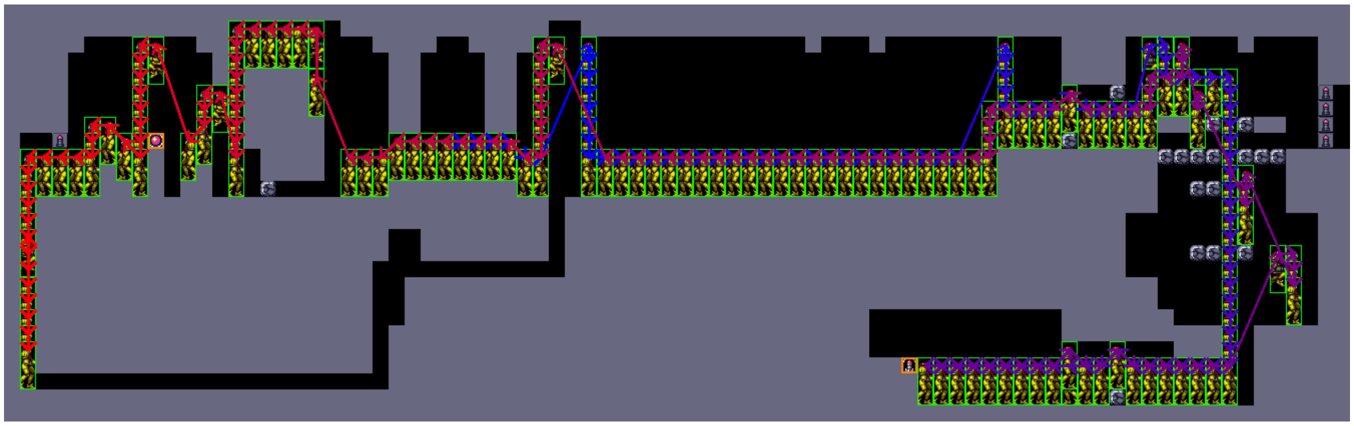

Figure 5: Example level 4(b) and counterexample trace output. The level satisfies $EF(goal)$ (the player can reach the goal from the initial state) but not $AG(EF(goal))$ (they can reach a state from which the goal is no longer reachable). Arrows are placed between consecutive player states. The color of an arrow signifies where it comes in the trace. Earlier transitions have blue arrows, shifting towards red later in the trace. This trace shows the player traveling to the right to pick up missiles, traveling back to the left and jumping over the morph ball power up, then falling into the pit and getting stuck. In 4(a) this is impossible because the player must pick up the morph ball power up before getting the missiles. Thus, even if they fall into the pit, the player can use the exit route to leave.

the Kripke structure is quickly assembled and checked using the `pyModelChecking` library.[7] In the example from Figure 4(a), the search enumerates 3645 states over a total of 933 total valid tile-level positions (positions where the player hitbox does not intersect a wall), for an average of 3.91 states per tile. In Figure 4(b), the search enumerates 4700 total states, or 5.03 states/tile. This increase in reachable states is due to newly-reachable states where the player has obtained missiles without the morph ball.

This process can scale well to larger levels. If the number of states per position and the number of transitions per state are constant, then the running time increases linearly with the number of new positions as the size of the level design increases. However, adding additional items might increase the number of states at each position exponentially. Verifying an entire game world with many items (and other additions to player state such as bosses defeated or keys) may still be out of reach.

5 GLOBAL STATE SPACE STRUCTURE

Since the Kripke structure represents the state space as a graph, we can use spectral embedding[8] [16] to visualize it as a two-dimensional diagram. Figures 7 and 6 show the structure of the level design in Figure 4(b). At the global scale, there are two broad paths from the initial state to the goal state. On one path, the player picks up the morph ball and then missiles. On the other, they pick up the missiles before morph ball, creating the potential for a softlock.

The structure of the embedded graph is locally almost one-dimensional, and movement is mostly either towards or away from item upgrade locations. This coincides with progress in the game being strongly tied to collecting spatially-separated items. Where the graph embedding shows branches, the player may make a choice

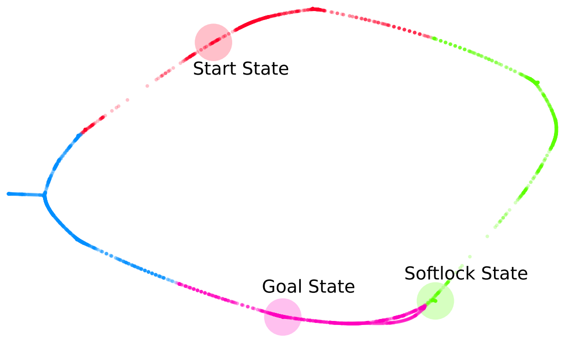

Figure 6: Spectral embedding, compare Figure 7.

about which segment to explore. For example, the player can enter the lower-left passage from the right side before collecting missiles. However, since the player cannot jump out, they have to leave the way they came. This gives rise to the short blue spur at the left of the embedding. In the same way, the small pink loop near the bottom of the figure exists because the player who has missiles and morph ball has two ways of traversing the left-side passageway.

In *Mario* it was convenient imagine that the state space was roughly two-dimensional. However, this view ignores some of the most salient structures in *Super Metroid*.

6 CONCLUSION AND FUTURE WORK

In this paper we used model checking to detect softlocks in *Super Metroid*. Underlying those specific contributions are broader ideas that should be considered in future research. The *playability* of a specific level design is more than just reachability of a goal location.

[7]https://github.com/albertocasagrande/pyModelChecking
[8]Specifically, we use `sklearn.manifold.SpectralEmbedding`.

	Search to Enumerate States	Adding Graph Edges	Checking $AG(EF(goal))$	Total
Figure 4(a)	3645 states in 3.24s (1125 states/sec)	15344 edges in 3.02s (5081 edges/sec)	0.45s (Property is True)	6.72s
Figure 4(b)	4700 states in 4.07s (1155 states/sec)	19625 edges in 3.99s (4919 edges/sec)	0.60s (Property is False)	8.67s

Table 1: Summary of wall-clock time used in the verification examples in Figure 4. Times are averaged over 5 trials.

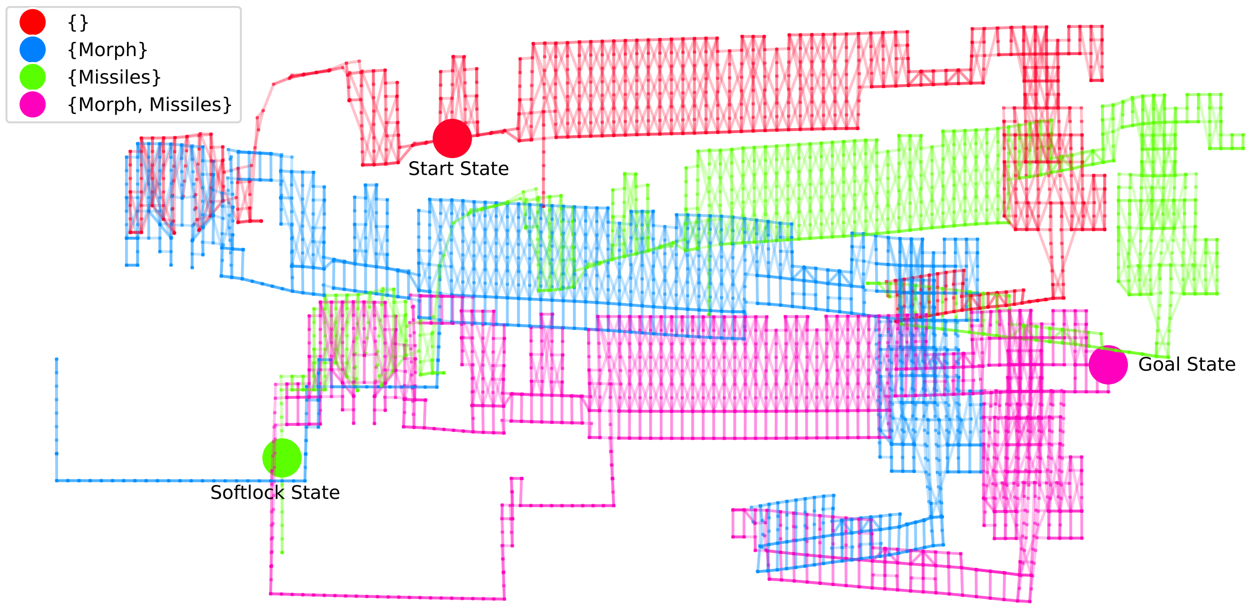

Figure 7: *Hybrid* spectral embedding of the level design in Figure 4(b). Each state is colored by which items the player has when in that state. Three important states are shown larger: The red state is the starting state, the pink state is the goal state, and the green state is the softlock state found in Figure 5. State positions are based on *both* the game world position and a spectral embedding of the state graph shown in Figure 6.

In *Mario*, many of these complex properties are hidden because of the simple gameplay mechanics. As mentioned earlier, a softlock in a *Mario* level will always be spatially near the design element that causes it. In order to generate playable levels for games with complex mechanics, we must peel back certain assumptions that apply only to *Mario* and *Mario*like games. This paper takes a key step in that direction by focusing on a game where all directions of movement are relevant, and consideration of game state beyond the obvious two-dimensional tile grid is critical for analyzing playability. Having no softlocks is just one example of the many critical or desirable properties of playable levels. Developing a rich set of formal properties that collectively express playability is an important direction for future work.

There are also a two main concrete unsolved problems in applying our approach to new games. One is the problem of creating an accurate movement model, the other is solving the problem of scaling. First, the movement model used for verification here was hand-authored, and may not fully describe the range of possible behaviors in the game. To ensure that the abstraction is sufficiently grounded in reality, this model should be validated by comparing it to actual gameplay. One way to do that would be to use the Reveal-More algorithm [1] together with a gameplay trace from the actual game to generate sequences of real states. By converting those real states to abstract states and checking whether the movement model is consistent with those movements, the abstract movement model could be validated. This method might also be used to generate a movement model from a gameplay trace. Game designers could then generate an accurate abstracted movement model for their game just by playing it, rather than carefully maintaining the hand authored abstraction as the design changes. The emerging literature on learning-based testing (LBT) offers some guidance in this direction [20].

Second, while our technique is efficient for small example levels, there are still too many possible states to allow practical verification of the entirety of *Super Metroid*. Scaling this idea to verify entire game worlds can be done by analyzing their individual components in a modular fashion. To do this we must efficiently combine sub-verifications for level components into a verification for the entire game. The idea of modular verification is well known in the formal methods literature [19], but it must be adapted to the distinct kind of modules relevant to level designers. It is not yet clear how to retrofit methods that combine proofs for local functions in software source code to combine proofs for local regions in a game world.

Finally, level design is a great domain for applying model synthesis techniques. Given a partially-designed or even a completely blank level, the unspecified parts can by *synthesized* while guaranteeing certain CTL properties over the entire space of play. This could be used to create a gameplay-aware level design assistant which guarantees more than simple goal reachability. Applying CTL techniques to mixed-initiative design could be used to create a system like Tanagra [23] for designing *Metroid*-like games. This system might be able to *suggest* the design in Figure 4(a) from a user-input 4(b).

All of these ideas stem from the concept that playable levels follow certain unspoken rules. By exploring the space of these principles, we can make level design and analysis systems that can reason about playability, and see hidden flaws in levels that could be missed by human designers.

REFERENCES

[1] K. Chang, B. Aytemiz, and A. M. Smith. 2019. Reveal-More: Amplifying Human Effort in Quality Assurance Testing Using Automated Exploration. In *2019 IEEE Conference on Games (CoG)*. https://doi.org/10.1109/CIG.2019.8848091

[2] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of Model Checking* (1st ed.). Springer Publishing Company, Incorporated. a. Section 2.2.1, p.30, b. Section 2.4.1 p.53, c. Section 13.3, p.394.

[3] M. Cook and A. Raad. 2019. Hyperstate Space Graphs for Automated Game Analysis. In *2019 IEEE Conference on Games (CoG)*. https://doi.org/10.1109/CIG.2019.8848026

[4] Seth Cooper and Anurag Sarkar. 2020. Pathfinding Agents for Platformer Level Repair. *Proceedings of the Experimental AI in Games (EXAG) Workshop at AIIDE* (2020).

[5] Sega Corporation. 1991. Sonic the Hedgehog.

[6] Fernando de Mesentier Silva, Scott Lee, Julian Togelius, and Andy Nealen. 2017. AI-Based Playtesting of Contemporary Board Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games* (Hyannis, Massachusetts) *(FDG '17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 10 pages. https://doi.org/10.1145/3102071.3102105

[7] Andrew Hoyt, Matthew Guzdial, Yalini Kumar, Gillian Smith, and Mark O Riedl. 2019. Integrating Automated Play in Level Co-Creation. *Proceedings of the Experimental AI in Games (EXAG) Workshop at AIIDE* (2019).

[8] Summerville Adam J. and Mateas Michael. 2016. Super Mario as a String: Platformer Level Generation Via LSTMs. In *Proceedings of the First International Joint Conference of DiGRA and FDG*. Digital Games Research Association and Society for the Advancement of the Science of Digital Games, Dundee, Scotland. http://www.digra.org/wp-content/uploads/digital-library/paper_129.pdf

[9] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. 2014. Monte Mario: Platforming with MCTS. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 293–300.

[10] Pearson Jordan. 2015. Why Artificial Intelligence Researchers Love 'Super Mario Bros.'. *Motherboard* (Oct 2015). https://www.vice.com/en/article/8q84zz/why-artificial-intelligence-researchers-love-super-mario-bros

[11] Nintendo Co. Ltd. 1983. Super Mario Bros.

[12] Nintendo Co. Ltd. 1986. Kid Icarus.

[13] Nintendo Co. Ltd. 1986. The Legend of Zelda.

[14] Nintendo Co. Ltd. 1994. Super Metroid.

[15] John McCarthy. 1990. Chess as the Drosophila of AI. In *Computers, chess, and cognition*. Springer, 227–237.

[16] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. 2001. On Spectral Clustering: Analysis and an algorithm. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*. MIT Press, 849–856.

[17] J. C. Osborn, Brian Lambrigger, and M. Mateas. 2017. HyPED: Modeling and Analyzing Action Games as Hybrid Systems. In *AIIDE*.

[18] Chris Pedersen, Julian Togelius, and Georgios N Yannakakis. 2009. Modeling Player Experience in Super Mario Bros. In *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 132–139.

[19] L. Pick, G. Fedyukovich, and A. Gupta. 2020. Automating Modular Verification of Secure Information Flow. In *2020 Formal Methods in Computer Aided Design (FMCAD)*. 158–168. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_23

[20] Muddassar Sindhu. 2013. *Algorithms and Tools for Learning-Based Testing of Reactive Systems*. Ph.D. Dissertation. KTH Royal Institute of Technology.

[21] Adam M. Smith and Eric Butler. 2013. Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design. In *In Proceedings of ACM Conference on Foundations of Digital Games*.

[22] A. M. Smith, M. J. Nelson, and M. Mateas. 2010. LUDOCORE: A Logical Game Engine for Modeling Videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. 91–98. https://doi.org/10.1109/ITW.2010.5593368

[23] Gillian Smith, Jim Whitehead, and Michael Mateas. 2011. Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on computational intelligence and AI in games* 3, 3 (2011), 201–215.

[24] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. 2010. The 2009 Mario AI Competition. In *IEEE Congress on Evolutionary Computation*. IEEE.

[25] Nathan Partlan Vivian Lee and Seth Cooper. 2020. Precomputing Player Movement in Platformers for Level Generation with Reachability Constraints. *Experimental AI in Games* (Oct. 2020). http://www.exag.org/papers/EXAG_2020_paper_13.pdf

[26] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 221–228.

[27] Zeping Zhan, Batu Aytemiz, and Adam M. Smith. 2018. Taking the Scenic Route: Automatic Exploration for Videogames. *CoRR* abs/1812.03125 (2018). arXiv:1812.03125 http://arxiv.org/abs/1812.03125

[28] Xiaoxuan Zhang, Zeping Zhan, Misha Holtz, and Adam M Smith. 2018. Crawling, Indexing, and Retrieving Moments in Videogames. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*.