Playspecs: Regular Expressions for Game Play Traces

Joseph C. Osborn and **Ben Samuel** and **Michael Mateas** and **Noah Wardrip-Fruin**
Expressive Intelligence Studio
University of California, Santa Cruz

Abstract

We introduce Playspecs, an application of ω-regular expressions to specifying play traces (sequences of game states or events unfolding over time). This connects the automated analysis and model checking of games to the literature on formal software verification via Büchi automata. We show how to define desirable or undesirable sequences of game events with Playspecs and how associated algorithms can find examples (or prove the impossibility) of such sequences. Playspecs have two main benefits over existing techniques for specifying the behaviors of a game over time. First, they offer a scalable commitment to formal modeling: the same Playspecs can filter existing traces gathered by telemetry, search for satisfying traces using existing game code, or drive formal verification when paired with a logical model of a game. Second, Playspecs' syntax can be customized for the game engine or game in question so designers may write specifications using their game's native vocabulary. We define Playspecs' syntax and semantics (modulo game-specific customizations) and outline algorithms for each of the applications mentioned above, providing examples from the social simulation game *Prom Week* and the puzzle game engine *PuzzleScript*.

1 Introduction

Game design produces complex emergent behaviors. Unfortunately, some of those behaviors may be undesirable, and some desirable behaviors may be absent: the design may not match the designer's intent. Furthermore, game *programs* do not always faithfully implement game *designs*. Techniques from the formal software verification community could help resolve both of these problems, but game designs are not always explicitly specified; even when specifications are explicit, they tend to be informal.

Informal or missing specifications require frequent playtesting and human interpretation to determine desirable or undesirable behaviors. Formal game modeling approaches like BIPED (Smith, Nelson, and Mateas 2009) automate some of this design verification, but often require a comprehensive logical model of the game design to operate. This can be challenging for game designers and programmers unfamiliar with formal logic. Even familiar tools like unit or functional tests have limited utility for checking the correctness of game programs: gameplay situations are complex to configure and correctness criteria often involve the evolution of game state over time. Automated testing of complex game functionality seems relatively uncommon in the game industry, as presentations at its flagship conference still advocate for it (Du Bois 2009; Provinciano 2015). Looking only at the behavior of games (rather than their code), most game play trace analysis uses aggregated metrics like event counts to collapse sequences of game states into sets of numbers (El-Nasr, Drachen, and Canossa 2013). Some work has also been done in searching for traces which contain certain events, in some cases only when prior to other events. Trace visualization and gestalt approaches (Liu et al. 2011; Osborn and Mateas 2014) are more prevalent than targeted queries; this may be because writing targeted queries is difficult, often combining multiple database lookups with code in a general-purpose programming language. We contrast our work with other solutions to these problems in Sec. 6.

We propose *Playspecs*, a formalism that cleans up and regularizes all of these trace analysis activities. We provide one play trace specification language (tailored for efficiently searching through play traces) that can be reused across different games. This language also scales up from trace filtering through to rigorous formal verification, permitting the use of the software engineering community's verification tools.

The input to these Playspecs can be witnessed play traces gathered from telemetry (or random play, or solution search) or a logical model of a game. Playspecs can be adopted at the level of the game engine or the individual game, and with a little developer effort they can be relatively natural for designers to author directly. Formally, Playspecs are ω-regular expressions over program states (instead of characters in a string); ω-regexes are a well-understood language for specifying the behavior of computational systems (Albin 2003). In introducing Playspecs, we contribute: an application of proven techniques from formal verification to game design; a specification tool with a scalable degree of formality; and an emphasis on the use of game- and game-engine-specific concepts and syntax when specifying a game design.

We have applied a subset of the Playspec language to *PuzzleScript* (Lavelle 2013), a game engine for designing puzzle games, to verify that level solutions are *valid* with respect to a designer's intent. Source code is available on GitHub in the

`js/analyzer/` directory of `https://github.com/joeosborn/puzzlescript/tree/analyzer` (though this implements an earlier draft of Playspecs). Given their promise in this relatively general application, we have also begun applying Playspecs to the analysis of play traces from the simulation game *Prom Week* (McCoy et al. 2011). We provide a relatively complete, efficient, and documented reference implementation at `https://github.com/joeosborn/playspecs-js`.

This paper proceeds as follows. First, we motivate the use of Playspecs with *PuzzleScript* and *Prom Week* examples. We then formally define the syntax and semantics of Playspecs for an audience that may be unfamiliar with existing work in program verification. Next, we outline how to apply Playspecs to existing games and game engines. Finally, we compare Playspecs with earlier approaches to the formal specification and verification of game designs.

2 Motivating Examples

PuzzleScript has already received some attention from the game AI community (Lim and Harrell 2014). It offers a semi-declarative syntax for defining 2D puzzle games in terms of *1-dimensional rewrite rules* which each match a slice of the current game *level* (a 2D grid of *objects*, where multiple objects may reside at the same position) and modify that slice according to a fixed pattern. For example, to detect a player object moving towards an adjacent box object (in any direction), one writes `[> Player | Box]`; to cause the box to move under this circumstance, the complete *rule* is `[> Player | Box] -> [> Player | > Box]` (note the > arrow which is added to the box on the right hand side). Levels are written out as 2D grids of ASCII characters whose meanings are assigned by a *legend*. Each level is won when the game's *win conditions*, e.g. `all Box on Target`, are met.

In puzzle games, it is important to gradually present the game's concepts to the player, building up to more complex problems. If a player can complete a stage without learning a necessary concept (for example due to an oversight in the level design) they may be unprepared for future levels. The designers of the math puzzle game *Refraction* invented means to detect and prevent these kinds of design bugs (Smith, Butler, and Popovic 2013), but applying their techniques to new games requires working largely from scratch.

We have extended *PuzzleScript* with an automated solver based on heuristic search and with support for Playspecs: any *PuzzleScript* game can define Playspecs that must hold for found solutions of each level. Solutions which violate those specifications are presented to the designer. In this way, we build confidence that levels have no shortcuts and require the player to have learned certain concepts. We simplify writing Playspecs by supporting a subset of *PuzzleScript*'s native syntax in the specification language. A specification can demand that a 1D pattern or 2D level fragment matches the current game state or that a win-condition-like predicate holds. These forms exactly mirror the *PuzzleScript* syntax.

Some solution checks expressible in Playspecs include "the red, green, and blue switches must be flipped in that order"; "the player must remove a box which starts on a goal position, replacing it later"; and "this level requires at least 20 moves, moving two particular boxes at least once each."

Playspec	ω-Regex	Explanation
`p`	`p`	The fact `p` holds in the current state
`P&Q`		`P` and `Q` both hold in current state
`P|Q`	`[pq]`	Either `P` or `Q` holds in current state
`not P`	`[^p]`	`P` does not hold in the current state
`F,G`	`FG`	Sequence `F` and then `G`
`F;G`	`F|G`	Either sequence `F` or `G` holds
`F^G`		Sequences `F` and `G` both hold
`...`	`.*`	Matches any number of states
`F ...`	`F*`	`F` matches zero or more states
`F 1...`	`F+`	`F` matches one or more states
`F M...N`	`F{M,N}`	`F` matches between `M` and `N` states
`..`	`.*?`	Reluctantly matches any number of states; same variations as `...`
`F***`	F^{ω}	`F` repeats forever

Figure 1: Playspec and analogous ω-regex syntax.

Prom Week is a social simulation puzzle game (driven by the AI system *Comme il Faut* (McCoy et al. 2014)) populated by characters who have ever-shifting *relationships* and *attitudes* towards each other. Players change the *social state* by having characters engage in *social exchanges* with each other, such as *Ask Out* or *Backstab*. Each level of *Prom Week* asks the player to solve different social puzzles such as wooing a date or giving a bully their just desserts within a limited number of turns. The exchanges available on a given turn are determined by over 5,000 *influence rules* which reason over all aspects of the social state, including relationships (friends, dating, enemies), network values (buddy, romance, cool), and other factors. This rich dynamism comes at the cost of predictability: though adding additional rules is important for making characters believable, it is difficult for the designers to anticipate how any given rule affects the rest of the system.

There are three main applications of Playspecs to *Prom Week*: finding traces with interesting or surprising behavior or strategies; verifying that level-specific goals can still be met when influence rules are changed or added; and identifying traces as *unbelievable* if characters perform unrealistic actions like repeatedly breaking up and starting to date again on successive turns. Playspecs can reason over the same social facts as the influence rules do with a custom syntax resembling labeled edges. For example, `Doug-friends-Jordan` checks whether Doug and Jordan have the `friends` relationship, and `Doug-romance<0.3-Chloe` succeeds if the value of Doug's romance network for Chloe is less than 0.3.

3 Playspecs

In this paper, we use the term *play trace* to mean a sequence of data generated by activity in a game. The simplest possible play trace might be a log of the inputs provided by the user. Richer traces carry more data: more abstract inputs and events, more elements of the game state, and so on. These traces might be physical files stored on disk or may be generated on the fly by a verification tool. In the case of *PuzzleScript*, a trace is a sequence of player movements

Puzzlescript Examples

`[ROff] & [GOff] & [BOff] 1..., [ROn], [GOff] & [BOff] 1..., [GOn], [BOff] 1..., [BOn], ...`

The red, green, and blue switches must be flipped in that order. Switch objects are named by a color (`R`/`G`/`B`) and status (`On`/`Off`) pair.

```
..., '=@', ..., '=*.=', ..., '=*.=', ..., '=P.=', ..., win & end
                =O..          =OP*          =@.*
                =..*
```

The player must move a box which starts on a goal position, replacing it after moving a second box. In these 2D patterns, `@` means both a box (`*`) and a target (`O`) are present; = indicates a wall and `P` the player. In this level, these patterns identify unambiguous locations.

`20... ^ ..., 5@5 [no Box], ... ^ ..., 7@3 [no Box], ...`

This level requires at least 20 moves, moving two particular boxes at least once each. `X@Y` forces patterns to match at a specific location.

Prom Week Examples

`Doug-romance<0.3-Chloe ... ^ not Doug-dating-Chloe 1..., Doug-dating-Chloe`

With a romance network value of less than 0.3 the entire time, Doug must go from not dating to dating Chloe.

`Doug-dating-Chloe, ..., not Doug-dating-Chloe, ..., Doug-dating-Jordan & Doug-friends-Chloe`

Doug begins by dating Chloe, but they eventually break up. Doug begins dating Jordan and befriends his old flame.

`Doug-friends-Chloe, ..., Doug-dating-Chloe ; Doug-dating-Chloe, ..., Doug-friends-Chloe`

Doug befriends Chloe and then starts dating her, or else Doug starts dating Chloe before they become friends.

Figure 2: Example *PuzzleScript* and *Prom Week* Playspecs

and level states found during a search for puzzle solutions. In *Prom Week*, traces contain individual social games and their outcomes; they come from either observed game play (gathered by telemetry) or from exhaustive enumeration of games up to a fixed number of turns. Playspecs assume that game traces are *sequences of sets of facts*.

A Playspec *matches* a trace in the same way that a regular expression matches a string. One regex might match several distinct positions in the string or use start and end anchors to only match complete strings; analogously, a Playspec may match a portion of a play trace or an entire trace. Just as a regex describes a set of possible strings (a *language*), each Playspec describes a set of possible traces.

A regex checks whether each character in the string is a member of a particular set of characters, but Playspecs support a variety of complex (often game- or game-engine-specific) queries on individual states. Boolean combinations of these game-specific *basic facts* make up the *state* language fragment of Playspecs, and the game-independent syntax for describing the evolution of states over time is the *trace* (or *regular*) language fragment (Fig. 1 describes both fragments in a side by side comparison of Playspec and ω-regex syntax).

There are four basic facts which are the same across all games: `true`, `false`, `start`, and `end`. The first two are self-explanatory; the third and fourth indicate the beginning and end of the play trace respectively (roughly analogous to regex `^` and `$` anchors). Note that while the propositional formulae of the state language can be negated, the temporal sequences of the trace language may not be; the lookaround extension we propose later could be put to this purpose.

All other basic facts are game- or genre-specific: Sec. 2 discussed how *Prom Week*'s basic facts query the social state, while *Puzzlescript*'s basic facts concern the current configuration of the puzzle. Some facts could be provided by general-purpose game engines; there is also a connection to the *authorial affordances* of operational logics and domain models (Mateas and Wardrip-Fruin 2009; Osborn et al. 2015), which are portable across game genres.

When considering a sequence of states in Playspecs, the fundamental requirement is a way to advance (or consume) the current state. Regular expressions implicitly advance the stream of characters: `abc` means an `a` followed by a `b` followed by a `c`. This can also be read as the regex `a` followed by the regex `bc`. This is called *concatenation*. Playspecs have more complex syntax for each individual state so we use a comma (`,`) to indicate concatenation; it could be read as *and then* or *followed by*. A simple example using *Prom Week*: the Playspec `Doug-mean-Chloe, Doug-guilty, Doug-nice-Chloe` would only match traces in which the player had Doug do something mean to Chloe, then made him feel guilty, and then had him be nice to her. Each of these three basic facts is checked in turn against successive positions of the trace.

A designer doesn't always know in advance how long a property should hold. For instance, we might want to find traces where two characters begin dating but eventually break up. This is analogous in regex to asking for a lowercase letter eventually followed by a number using the Kleene star (`[a-z].*[0-9]`); there can be any number of characters between our two points of interest

PuzzleScript Playtraces

Original Trace	Sequence of Sets of Facts
{start_state: '======= =.....= =O*..P= =.....= =======', moves:['left',	turn = 1, move = left, ..., layer0 @ (1,2) = background, layer1 @ (1,2) = target, layer0 @ (2,2) = background, layer2 @ (2,2) = box, layer0 @ (5,2) = background, layer2 @ (5,2) = player, ...
'left',	turn = 2, move = left, ..., layer0 @ (4,2) = background, layer2 @ (4,2) = player, layer0 @ (5,2) = background, layer0 @ (6,2) = background, ...
'left'] }	turn = 3, move = left, winning, ..., layer0 @ (1,2) = background, layer1 @ (1,2) = target, layer2 @ (1,2) = box, layer0 @ (2,2) = background, layer0 @ (2,2) = player, ...

Prom Week Playtraces

Original Trace	Sequence of Sets of Facts
<SocialGameContext gameName='Share Interest' initiator='Doug' responder='Jordan' effectID='8' chosenItemCKB='retro phone'/>	time = 1, doug-!share_interest(8, 'retro phone')-jordan, chloe-cool=0.7-doug, chloe-romance=0.8-doug, chloe-friend=0.5-doug, ...
<SocialGameContext gameName='Pick-Up Line' initiator='Chloe' responder='Doug' effectID='6'> <SFDBLabel type='funny' from='Chloe' to='Doug'/> <SFDBLabel type='romantic' from='Chloe' to='Doug'/> </SocialGameContext> ...	time = 2, chloe-!pick_up_line(6)-doug, chloe-funny-doug, chloe-romantic-doug, chloe-cool=0.7-doug, chloe-romance=0.9-doug, chloe-friend=0.6-doug, ...
<SocialGameContext gameName='Reminisce' initiator='Doug' responder='Chloe' effectID='8' other='Jordan'/>	time = 16, doug-!reminisce(8, jordan)-chloe, ...

Figure 3: Play trace data from a simple *PuzzleScript* game and *Prom Week*. For each game the left column illustrates how play traces are recorded, while the right shows (abstractly) the information made available for Playspecs to query at each timestep.

(the letter and the number), just as any sequence of game states or player actions may transpire between the characters falling in and out of love. In Playspecs, this *repetition* might be written as `Doug-dating-Chloe 1..., not Doug-dating-Chloe`. The use of `...` is read as *dating until no longer dating* (the preceding numeral `1` requires that the characters are dating for at least one state). The ellipses describe potentially multiple states in which a property holds (`Doug-dating-Chloe` in our example), with `true` as a default. `...` is also *greedy*: it will prefer to consume as many states as possible when matching. The `..` variants consume as few states as possible. When there are multiple matches, these operators will yield all the same matches in opposite order. Minimum and maximum bounds can also be provided (as in `1...`), and these default to 0 and infinity.

The regex notion of *alternation* can be used to discover if at least one of several Playspecs holds. We can look for either an `a` followed by a `b` or else `xyz` using the regex `ab|xyz`. To avoid ambiguity with the state language's propositional disjunction `|`, and for symmetry with the `,` of concatenation, Playspecs use `;` for alternation. Continuing with the dating example, we might want traces where Doug ends up single, i.e., he breaks up with whoever he dates or else never dates in the first place. The Playspec `not Doug-dating-Chloe ...; Doug-dating-Chloe 1..., not Doug-dating-Chloe` matches when Doug never dates Chloe or they date before eventually breaking up, but doesn't account for Doug and Chloe resuming their relationship and staying together afterwards. By applying the repetition operator to that whole specification, we can match only traces in which Doug never lives happily ever after: `start, (not Doug-dating-Chloe...; Doug-dating-Chloe 1..., not Doug-dating-Chloe)..., end`.

Note that the number of states consumed by the alternation depends on the branch that matched. A Playspec like `Doug-lonely, (Doug-dating-Chloe; Doug-friends-Oswald, Doug-friends-Jordan), not Doug-lonely` consumes either three or four states. Alternation can be understood as cloning the expression once for each operand (i.e., either side of the `;`), replacing the expression with each of the operands, and succeeding if any of these clones match.

As a notational convenience, we introduce `^`, read as *and* or *intersection*, which is dual to `;`. Like `;`, it effectively clones the Playspec for each operand. Unlike `;`, all clones must match the *same* portion of the trace for the match as a whole to succeed. For simplicity we require that all operands consume the same fixed number of states or that they can be stretched via `...` to fit the same segment of trace. Formally, this is the intersection of languages: a grammar which only matches strings that appear in both languages. This syntax can be used to find play traces where multiple paths leading to a single state must contain certain events or state sequences. Suppose we want to know when Doug becomes enemies with his former friend *and* lover, regardless of whether their initial relationship was Platonic or romantic: `(..., Doug-friends-Chloe, ... ^ ..., Doug-dating-Chloe, ...), Doug-enemies-Chloe`. While regex libraries in most programming languages do not offer intersection, we include it, in part to help define universal quantification (an extension we leave for future work). Further examples of the Playspec syntax described thus far can be found in Fig. 2.

So far, our syntax only recognizes play traces of finite length. While not applicable for *matching* existing traces, Playspecs which specify traces of infinite length can be useful

for *verifying* game designs. Checking for infinite loops can detect cases where players get stuck, which would only show up as quitting in a real trace. To forbid such infinite traces, we must be able to describe them. We therefore assume by default that Playspecs recognize finite portions of traces, but introduce syntax for recognizing an infinite *suffix* which satisfies a Playspec repeatedly. The *forever* operator (called ω in ω-regex) is written with $\star\star\star$: it is a semantically and visually lifted version of `...` indicating that the Playspec it modifies repeats forever. It may only appear at the end of a Playspec, and it will not match any finite play trace. Unlike `...`, it accepts no time-bounding arguments.

4 Integration with Existing Games

There are two main decisions when integrating Playspecs with an existing game or game engine: the degree of formality and the content of traces. For *Prom Week*, we use Playspecs to filter and match existing play traces; this requires the least effort but is also the least flexible. With *PuzzleScript*, we generate solutions using heuristic search and then test those solutions against Playspecs; this requires few modifications to the underlying game engine, but exhaustively checking solutions has a high computational cost. Formally modeling the game under consideration would require extra authoring effort but could answer targeted queries very quickly.

The first step towards integrating Playspecs is transforming the game's play traces into sequences of sets of facts. This could involve modifying how traces are recorded, converting traces in an external tool, or performing on-demand translation into this format. In *PuzzleScript* we start with a sequence of input directions which we replay through the game engine to recover the configuration of the level at each step. We write Playspecs over these augmented traces. A game with more complex state might define each set of facts implicitly via a function that determines the truth of a proposition.

Fig. 3 shows examples of play traces both in game-native formats and after augmentation with extra state data from re-simulating the recorded input sequence. The concrete syntax shown is only for illustration: for example, each "set of facts" in the *PuzzleScript* traces is stored as an array of integers.

As for what comprises a trace, there are three important factors. First is the trace's level of abstraction: the game activity represented in the trace. A trace might contain frame-by-frame or turn-by-turn game states or abstract level-by-level progression. Frame-by-frame traces will be too fine-grained for most use cases besides the unit-testing application.

Second is whether the trace contains instantaneous events, durative states, or both. Using only game events will yield more compact traces, but some specifications may be harder to write; on the other hand, traces with only states may make other Playspecs awkward. Including both (or recovering states by replaying inputs) can be a good option.

Finally, the implementer must decide on a syntax for game-specific basic facts. These often use tokens and syntactic structures outside of the existing Playspec grammar, for example *Prom Week's* relationship tests or *PuzzleScript's* 1D patterns. Parsing and checking these predicates is necessarily implementation-dependent, but we believe a portable grammar formalism could cover most use cases; due to space limitations we leave that for future work. Our JavaScript reference implementation instead provides a customizable parser. One could also imagine a generic fact syntax.

5 Checking Playspecs

We now outline algorithms and applications of existing tools to check Playspecs. The simplest case is matching a Playspec against a specific witnessed trace. As with regular expressions, Playspecs are efficiently matched by mechanical transformation into finite automata or an equivalent representation (Thompson 1968; Cox 2011). Playspecs using the $\star\star\star$ operator may be rejected in this application—any real play trace is necessarily finite. The main difference from regex matching algorithms is that instead of checking a stream of bytes against values or ranges of values, an implementation checks a sequence of states against logical formulae over each state's set of facts. If a specific game already supports more formal usage of Playspecs, that work can be reused to decide whether an observed trace meets a Playspec: for example, a trivial formal model could be synthesized which only advances through the states in a given trace.

The second level of formality pairs Playspecs with search. For *PuzzleScript*, we developed a heuristic search which solves levels and then ensures those solutions match given Playspecs. We currently only match against complete solutions, but it would be straightforward to match each candidate action incrementally to guide search towards solutions that violate the specifications. Integrating Playspecs with the *PuzzleScript* engine required minimal additions beyond the reloading and replaying necessary for the automated solver's operation; parsing and evaluating the basic facts largely reused built-in features of the engine.

Finally, we consider the verification of Playspecs given a formal model of the game design. Program verification tools commonly work by transforming both the input program (often in a special-purpose modeling language) and the negation of a logical specification into Büchi automata (the infinite analogue to finite automata), intersecting them, and finding whether there is any sequence of inputs accepted by the newly constructed automaton (Courcoubetis et al. 1993). This deserves unpacking. *Büchi automata* are used instead of the regex-equivalent finite state automata because programs running in finite memory have finitely many states, but may loop infinitely. We use the *negation of the specification* to find whether a trace can be produced which *violates* the original specification, in other words one which satisfies its negation. *Intersecting two automata* means producing a third which only accepts those traces which both the originals would accept—those traces that the program could conceivably produce which also satisfy the negation of the specification. Finally, *detecting non-emptiness* of the language represented by the resulting automaton tells us whether there exist problematic traces. This last check is computationally easy once all the other work has been done. All this is to illustrate that Playspecs can be checked by standard algorithms and, indeed, by many standard tools, since they are readily translated to conventional ω-regular expressions and thence to Büchi automata (Holzmann 1997; Cimatti et al. 2000; Duret-Lutz and Poitrenaud 2004). The

Playspec and the formal model should be at the same level of abstraction here—for instance, if the Playspec concerns game turns rather than frames, the model should as well. In practice, formal models could certainly be authored by hand using formalisms like transition systems, event calculus, Machinations diagrams (Dormans 2009), or a partial order of requirements (Van der Linden 2013). Existing non-game modeling languages like Promela (Holzmann 1993) are also excellent candidates, and the translation from Playspecs to a format which those tools support is straightforward and could be automated. If the implementer wants to work directly against their game's program code, semi-automated tools like SPOT can transform a conventional program into an automaton with some API support (Duret-Lutz and Poitrenaud 2004). There are also program verification tools which can recover models from programs without such API help at the cost of larger models, e.g. DiVinE (Barnat et al. 2013).

Finally, it is worth noting that there are many extensions to both linear temporal logic (a conventional verification language which is translated into the Büchi automata described above) and regexes, ripe for inclusion into Playspecs. Some examples include Metric-LTL (Koymans 1990), which is appropriate for realtime systems like action games; regex lookaround, which could make phrasing certain properties much more concise; first-order quantification, which permits a high level of abstraction over traces — in short, Playspecs that don't check the state of specific characters (like Doug or Oswald) but rather the states of *roles* that could be filled in by any character ("Doug's friend", "Oswald's lover") (Kröger and Merz 2008); Probabilistic-LTL which describes likely versus unlikely possibilities (Baier 1998); and regex-style capture groups to identify particular subsequences of interest within the matched segment of a play trace (our reference implementation supports this extension already).

6 Related Work

Playspecs are strongly related to previous logical formalisms that have been proposed for specifying game traces. The most similar is probably the trace grammar proposed for *Ludocore* (Smith and Mateas 2011). This formalism queries existing traces with arbitrarily quantified Boolean formulae using a Prolog-like syntax, a subset of which can be used with a logical game engine to perform targeted search. Compared to the high computational complexity of this approach, Playspecs with their linear-time, backtracking-free match-query procedure fill a potentially large niche for which the prior work is inappropriate. Furthermore, Playspecs integrate with existing program verification tools and the regular expression-like syntax may be easier for game designers to author than the Prolog-like syntax of the prior work.

In general, techniques which permit the formulation and checking of useful properties over game traces require a total investment in formal methods, including logically modeling much of the game in question (Smith, Nelson, and Mateas 2009; Shaker, Shaker, and Togelius 2013; Butler et al. 2013). Playspecs offer a scalable investment in formality: it should be possible to write Playspecs over existing traces with relatively little effort to determine whether Playspecs will be of use to the designer. Further integration with a game engine or formal game modeling can be approached piecemeal and use the exact same specifications. One could even use Playspecs as a frontend, reducing them to sets of first-order logic event calculus statements.

These latter tools are often referred to as automated playtesters; besides such player-centric testing, Playspecs can support the unit-testing activities of game programmers. In this sense, they could be viewed as a generalization of *Inform 7's* Skein (Nelson 2006): instead of expecting a concrete output text for a given input sequence, a tester could provide inputs and require that, on replay, the resulting trace satisfies one or several Playspecs. Playspecs' ability to elide details about the sequence of game states should make writing unit tests for games easier and more modular. If the input sequence were also generated via Playspecs, property-based testing tools (Arts et al. 2006; Hughes, Norell, and Sautret 2010) could find violations; this is an incomplete solution compared to formal verification, but might be easier to integrate with existing software.

Finally, we note that some effort has been expended on converting puzzle solutions into strings where each character encodes information about a solution step (Andersen, Gulwani, and Popovic 2013). We have learned via personal communication that researchers have analyzed such strings using textual regular expressions; this is similar to our method in that it uses regular expressions, but it is less general.

7 Conclusion

We have described Playspecs, motivations for their use, ways to integrate them into existing games and game engines, and related work in the field of games. Implementers can use this paper and our reference implementation as a starting point.

We hope that game developers—especially game engine developers—adopt Playspecs at least at the level of matching and selecting existing or randomly generated traces, and ideally that they offer ways to drive their game engines via Playspecs. We further hope that game researchers will consider Playspecs as their language of choice for specifying properties of interest for formal models of games. We believe that as a language for defining the evolution of game states over time, Playspecs could undergird a rigorous study of game *dynamics* in the sense used by the MDA framework (Hunicke, LeBlanc, and Zubek 2004), which are so far under-theorized compared to game mechanics and aesthetics. Playspecs could be applied to AI play by running them *backwards*, generating rather than recognizing traces (as in the use of linear temporal logic in the planning community (Baier and McIlraith 2006)). We also suspect that Playspecs (perhaps with fuzzy or probabilistic extensions) have applications for general game playing, particularly in opponent modeling or characterizing sets of play traces produced by random or self-play. They could also be used in generative systems to filter out generated content that violates designer-specified invariants, or to support player-adaptive games by matching against the currently-unfolding play trace (perhaps in the service of drama management).

With extensions for first-order quantification, Playspecs could describe game rules directly; the authors have done so for noughts-and-crosses and other simple games. The insight

is that a game implicitly defines a set of valid input traces, and Playspecs also define sets of traces. Game rules whose effects are distributed over time or with complex conditions are especially good candidates for definition in this style.

Playspecs' utility is determined in large part by the naturalness of their syntax. Making this syntax convenient enough to define on a game-by-game basis is key to their success and represents important future work.

References

Albin, K. e. a. 2003. Property specification language reference manual.

Andersen, E.; Gulwani, S.; and Popovic, Z. 2013. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 773–782. ACM.

Arts, T.; Hughes, J.; Johansson, J.; and Wiger, U. 2006. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, 2–10. New York, NY, USA: ACM.

Baier, J. A., and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, 788. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Baier, C. 1998. On algorithmic verification methods for probabilistic systems. *Universität Mannheim*.

Barnat, J.; Brim, L.; Havel, V.; Havlíček, J.; Kriho, J.; Lenčo, M.; Ročkai, P.; Štill, V.; and Weiser, J. 2013. Divine 3.0–an explicit-state model checker for multithreaded c & c++ programs. In *Computer Aided Verification*, 863–868. Springer.

Butler, E.; Smith, A. M.; Liu, Y.-E.; and Popovic, Z. 2013. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 377–386. ACM.

Cimatti, A.; Clarke, E.; Giunchiglia, F.; and Roveri, M. 2000. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2(4):410–425.

Courcoubetis, C.; Vardi, M.; Wolper, P.; and Yannakakis, M. 1993. Memory-efficient algorithms for the verification of temporal properties. In *Computer-aided Verification*, 129–142. Springer.

Cox, R. 2011. Implementing regular expressions. https://swtch.com/ rsc/regexp/.

Dormans, J. 2009. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, 33–40.

Du Bois, P. 2009. Robotic testing to the rescue. In *Game Developers Conference*.

Duret-Lutz, A., and Poitrenaud, D. 2004. Spot: an extensible model checking library using transition-based generalized büchi automata. In *Proceedings of the IEEE Computer Society's 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 76–83. IEEE.

El-Nasr, M. S.; Drachen, A.; and Canossa, A. 2013. *Game analytics: Maximizing the value of player data*. Springer Science & Business Media.

Holzmann, G. J. 1993. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems* 25(9):981–1017.

Holzmann, G. J. 1997. The model checker spin. *IEEE Transactions on software engineering* 23(5):279–295.

Hughes, J.; Norell, U.; and Sautret, J. 2010. Using temporal relations to specify and test an instant messaging server. In *Proceedings of the 5th Workshop on Automation of Software Testing*, 95–102. ACM.

Hunicke, R.; LeBlanc, M.; and Zubek, R. 2004. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4.

Koymans, R. 1990. Specifying real-time properties with metric temporal logic. *Real-time systems* 2(4):255–299.

Kröger, F., and Merz, S. 2008. First-order linear temporal logic. In *Temporal Logic and State Systems*. Springer. 153–179.

Lavelle, S. 2013. Puzzlescript. http://puzzlescript.net.

Lim, C.-U., and Harrell, D. F. 2014. An approach to general videogame evaluation and automatic generation using a description language. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Liu, Y.-E.; Andersen, E.; Snider, R.; Cooper, S.; and Popović, Z. 2011. Feature-based projections for effective playtrace analysis. In *Proceedings of the Sixth International Conference on the Foundations of Digital Games*, 69–76. ACM.

Mateas, M., and Wardrip-Fruin, N. 2009. Defining operational logics. *Digital Games Research Association (DiGRA)*.

McCoy, J.; Treanor, M.; Samuel, B.; Mateas, M.; and Wardrip-Fruin, N. 2011. Prom week: social physics as gameplay. In *Proceedings of the Sixth International Conference on the Foundations of Digital Games*, 319–321. ACM.

McCoy, J.; Treanor, M.; Samuel, B.; Reed, A.; Mateas, M.; and Wardrip-Fruin, N. 2014. Social story worlds with comme il faut. *IEEE Transactions on Computational Intelligence and AI in Games PP (99)* 1–1.

Nelson, G. 2006. Natural language, semantic analysis, and interactive fiction. *IF Theory Reader* 141.

Osborn, J. C., and Mateas, M. 2014. A game-independent play trace dissimilarity metric. *Proceedings of the Ninth International Conference on the Foundations of Digital Games*.

Osborn, J. C.; Lederle-Ensign, D.; Wardrip-Fruin, N.; and Mateas, M. 2015. Combat in games. In *Proceedings of the Tenth International Conference on the Foundations of Digital Games*.

Provinciano, B. 2015. Automated testing and instant replays in retro city rampage. In *Game Developers Conference*.

Shaker, N.; Shaker, M.; and Togelius, J. 2013. Evolving playable content for cut the rope through a simulation-based approach. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Smith, A. M., and Mateas, M. 2011. Towards knowledge-oriented creativity support in game design. In *Proceedings of the 2nd International Conference on Computational Creativity*.

Smith, A. M.; Butler, E.; and Popovic, Z. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *Proceedings of the Eighth International Conference on the Foundations of Digital Games*, 221–228.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2009. Computational support for play testing game sketches. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Thompson, K. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11(6):419–422.

Van der Linden, R. 2013. *Designing procedurally generated levels*. Ph.D. Dissertation, TU Delft, Delft University of Technology.