s^{4}LVE: Shareable videogame analysis and visualization

Eric Kaltman
ekaltman@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania

Joseph C. Osborn
joseph.osborn@pomona.edu
Pomona College
Claremont, California

John Aycock
aycock@ucalgary.ca
University of Calgary
Calgary, Alberta, Canada

ABSTRACT

We describe a new browser-based tool for analyzing the behavior of computational systems, with worked examples for the Atari 2600. Our tool, s^{4}LVE (System State Sequence Search Language and Visualization Environment), consists of three main parts. First, we define a domain-specific visualization language tailored for understanding low-level memory operations. Second, we leverage a discrete time-series pattern matching language inspired by regular expressions to capture states and memory locations of interest. Third, we integrate these little languages with an intuitive, spreadsheet-based visual interface juxtaposed with a live emulator. This combined system supports both the incremental exploration of complex emergent systems and rapid iteration on new visualizations.

CCS CONCEPTS

• **Social and professional topics** → **History of software**; • **Human-centered computing** → *Visualization systems and tools*; • **Applied computing** → Computer games.

KEYWORDS

game studies, visualization, memory, emulation, domain-specific language

ACM Reference Format:
Eric Kaltman, Joseph C. Osborn, and John Aycock. 2019. s^{4}LVE: Shareable videogame analysis and visualization. In *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19), August 26–30, 2019, San Luis Obispo, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3337722.3341826

1 INTRODUCTION

Interactive software—including videogames—are highly emergent systems defined in terms of complex interactions between remote bits of program code. These programs exhibit their behavior through human-visible traces (like the pixels eventually drawn to the screen), transient digital storage (e.g., RAM or CPU registers), and implicit information at the level of game design knowledge (for example, maps of game worlds too large to fit in uncompressed form in memory). Even the data structures which fully exist within the machine are extremely transient—they might change in between drawing one row of screen pixels and the next!

Code | Sheet | Export | Vis

	A	B	C	D	E	F	G	H	I	J	K
0	Pitfall!	PCG	(LFSR at $81)								
1											
2	e2	=									
3											
4	1	1		1	0	0		0	1	0	
5											
6			1x hole	000						000	1x logs
7			3x holes	001						001	2x logs
8		tarpit	+ vine	010						010	2x logs
9		swamp	+ vine	011						011	3x logs
10			crocs	100						100	1x log
11		treasure +	black quicksand	101						101	3x logs
12		black quicksand	+ vine	110						110	fire
13		blue quicksand	no vine	111						111	snake
14											
15										x0x	no vine
16										x1x	vine
17											

Figure 1: A s^{4}LVE visualization of *Pitfall!*'s procedural level generation. (All interface figures are modified to show salient features.)

Whether we want to study the design elements of a videogame or the artistry behind its internal programming (or its copy protection scheme, or quirks of the underlying hardware), we generally have two available approaches as researchers: a deep dive and "reading" into the game's assembly code and compressed internal data, or the construction of automated tools which record and interrogate the game's runtime behavior (perhaps through instrumented game console emulators). We then take insights and data gleaned from these techniques and narrativize or visualize them in service of a particular argument.

These approaches have produced powerful and informative visualizations ranging from floppy disk and audio driver activity to

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

ACM ISBN 978-1-4503-7217-6/19/08.
https://doi.org/10.1145/3337722.3341826

illustrations of anti-piracy and maze generation code.[1] The resulting visualizations have functioned as informative snippets of system functionality, from floppy disk access and audio driver output to illustrations of anti-piracy and maze generation code. Unfortunately, combining deep code reading with ad hoc reverse engineering tools does not scale well as a research practice. An individual researcher cannot meaningfully examine the binary code of very many games within their career, and writing programs to analyze programs is hard (and in many cases potentially impossible thanks to the Halting Problem). Even worse, only those researchers who are both savvy games scholars *and* effective reverse engineers can read and manipulate game programs in this comprehensive way. Moreover, these researchers must also be talented graphic designers if they want to create effective visualizations to enable their work to reach the widest possible audience of readers.

We have synthesized insights drawn from reading and creating these types of hybrid procedural / visual arguments into a new interactive environment and language for doing systems-level game analysis. In Figure 1, we showcase our new web-based prototyping environment s^{4}lve (System State Sequence Search Language and Visualization Environment), which we believe helps to integrate and support the diverse skillset needed to analyze internal system design and behavior. s^{4}lve combines a novel domain-specific language (DSL) for visualizing game system state changes with the powerful sequence analysis language Playspecs [9] and integrates them into a convenient web-based development and visualization environment. Making these custom analyses as trivially shareable as web pages immediately makes them available not only for academic argument, but also for public education, library, and museum purposes; moreover, this makes them easy to integrate into pedagogy [1]. Shareability through a web interface also removes local dependencies which improves the reproducibility of s^{4}lve's visualizations, allowing them to be embedded into arguments in a shared context and modified by motivated readers. We believe this shared analysis context will allow for greater participation from a more diverse set of researchers and students, especially those who might lack the domain knowledge operationalized by s^{4}lve interface and DSL.

2 PROCESSING GAME PLAY STREAMS

The current version of s^{4}lve embeds the Atari 2600 emulator Javatari [10]. We selected it due to the importance of Atari 2600 games and to simplify embedding and instrumenting the emulator in our web-based environment. Hooks have been inserted both after each Atari Television Interface Adapter vertical blank and after each central processing unit instruction. These hooks are used to transfer control to s^{4}lve's interpreter. s^{4}lve consists of three integrated (but modular) components: *Playspecs*, which function like regular expressions for sequences of states and are used to describe and capture such sequences; the s^{4}lve DSL, which combines a specialization of Playspecs with a `lex`-like action language; and the spreadsheet-based visualization environment, which also supplies a vocabulary of actions to the DSL.

[1] Examples may be found at https://doi.org/10.11575/37x6-9690.

```
      start ::= { globalstmt } [ 'spreadsheet' BYTE* ] EOF
globalstmt ::= NEWLINE
globalstmt ::= STATE action
globalstmt ::= [ STATE ] PLAYSPEC action
     action ::= stmt
     action ::= '{' { stmt } '}'
       stmt ::= { WORD } NEWLINE
```

Figure 2: EBNF grammar for s^{4}lve's DSL.

2.1 Playspecs

Playspecs is a customizable generalization of regular expressions, a text processing scheme founded on the theory of finite automata and regular languages. The first generalization is to consider not just characters of strings, but arbitrary tokens describing (for example) game states. Specs written in the Playspecs syntax can therefore describe any regular grammar of game situations: the period of time a character stays at some position until they leave it, a player first achieving one goal and then some time later achieving another, a sequence of steps during which either of two Playspecs matches (or both of them, or one after the other), and so on. We refer the reader to earlier work on Playspecs for more examples [9].

The second generalization of Playspecs is to allow for arbitrary user-defined predicates to accept or reject such a state. Where a regular expression over strings might use a character class like `[abc]` to accept a letter which is in the set containing `a`, `b`, or `c`, a Playspec can use propositions of predicate logic.

In s^{4}lve, Playspecs are supplemented with two memory-reading predicates: `at:region@location(pattern)`, which succeeds when the memory region named by `region` contains, at the memory offset specified by `location`, a sequence of bytes matching `pattern`; and `changed:region@location`, which succeeds when the value at the given memory address *differs* from its previous value. Memory regions can, for example, be `ram` for main memory or `cpu` for CPU registers. Locations might be (for example) hexadecimal byte addresses or named registers.

Since propositions can be combined (or negated) using conventional logical operators, these two predicates together give an expressive declarative language for parsing low-level operations of videogame hardware. Playspecs also allows for a modular extension of this syntax to cover, for example, properties of dynamically allocated objects in a game engine, scene graph hierarchies, or vertex and index buffers sent to graphics cards.

2.2 s^{4}lve's Domain-Specific Language

A s^{4}lve state analysis specification is expressed in an imperative domain-specific language [8] which itself embeds the declarative Playspecs language. This DSL allows common game state patterns and their associated actions—and in particular, interaction with the spreadsheet—to be easily written without recourse to more general-purpose languages. The DSL also supports an "escape" mechanism that allows arbitrary JavaScript code to be run if needed for more advanced applications.

The DSL grammar is shown in Figure 2. As a language associating patterns with actions, it seemed natural to base its high-level design

begin	change to a different matching state
continue	continue matching subsequent patterns
frame	switch to frame-by-frame time steps
instr	switch to instruction-by-instruction time steps
eval	evaluate an arbitrary JavaScript expression
log	log a message to the browser's JavaScript console
message	display a message in the UI's text box
bubble	display a pop-up message at a spreadsheet location
highlight	highlight a range of the spreadsheet
label	change a spreadsheet label or range of labels
normal	un-highlight a range of the spreadsheet

Figure 3: s4LVE's *general, browser,* and *spreadsheet* actions.

on that of `lex` [5]. In its simplest form, a pattern-action pair fits on one line:

```
at:cpu@PC(f000)       log "start address"
```

This pattern, a single-state Playspec which matches when the CPU's PC (program counter) register has the value `0xf000`, would `log` a message to the console. Logging a message is only one of many possible actions; the full set is given in Figure 3. Multiple actions can also be specified for a single pattern.

Patterns can be qualified by *states*, and a qualified pattern will only be considered if the DSL interpreter is currently in that state. For example, here the above pattern-action pair is preceded by the state name `foo`:

```
<foo> at:cpu@PC(f000)       log "start address"
```

The `begin` action can be used to change the current state, e.g., `begin foo`, or `begin ""` to return to the original start state. Actions can be executed immediately when a DSL specification is loaded by using the special `START` state.

To handle the common case where a set of s4LVE actions is tightly coupled to a particular *spreadsheet*, the specification may optionally end with the `spreadsheet` keyword. Everything after that point is treated as a spreadsheet specification that is loaded accordingly.

2.3 Spreadsheet Interface

The current version of s4LVE commits to a spreadsheet-style interface for visualizing game state sequences. This is not an intrinsic limitation, but a specific example from which future interfaces could be generalized. s4LVE's visualization interface (shown in Figure 1) is split between a running Javatari instance and separate tabs devoted to (1) s4LVE code entry, (2) the spreadsheet specification form (spreadsheet), (3) an active heatmap display of the Atari's 128 bytes of RAM (`0x80–0xFF`), and (4) a JSON import-export for the spreadsheet. Of these features, the most germane is the spreadsheet.

Commands executed by the DSL, such as highlighting, changing a value, or providing a "bubble" message overlay based on memory events, are reflected in the spreadsheet (see Figure 3). An analyst can also define spreadsheet cells by reference to functions linked to the underlying runtime state. For instance, inserting a runtime location, like `at:ram@0x89` or `at:CPU@A`, into a cell will bind the location's value to the cell's displayed value and update accordingly. Furthermore, other cell functions can then alter a cell's CSS style

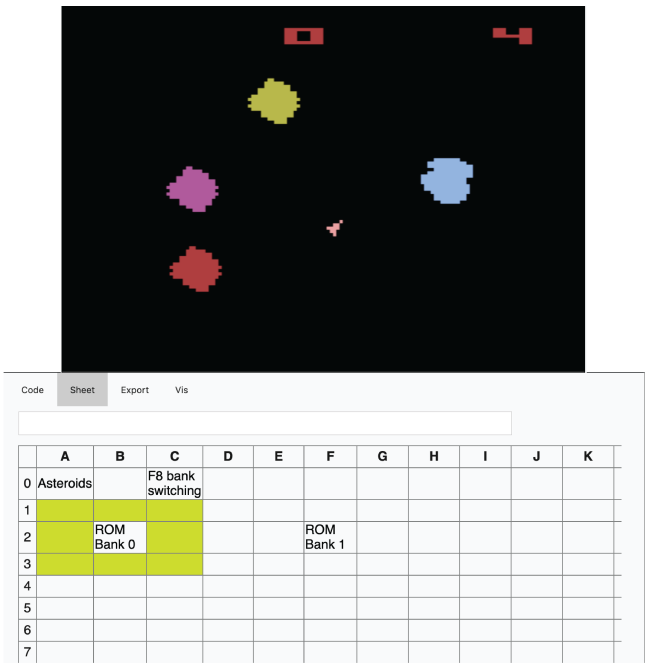


Figure 4: *Asteroids*'s bank switching visualized with s4LVE.

based on a location's value, `heat(<location>,<color>)`, or based on the cell's computed value, `eval(<cell_value>)`.

The import-export feature uses a simple JSON key / value format — `cell_coordinate:expression` — to allow other programs, including the DSL, to alter cell values and functions. This avoids potentially laborious manual entry in the spreadsheet itself if larger chunks of memory need to be visualized.

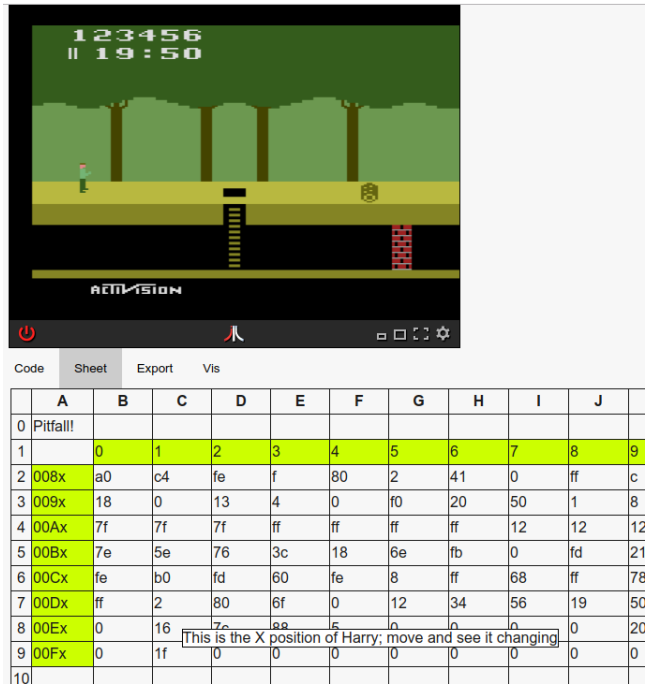
3 DEMONSTRATION

In this section, we present three worked examples illustrating typical uses of s4LVE: An interactive exploration of *Asteroids* ROM bank-switching, an explication of how *Pitfall!* encodes the entire game screen contents in a single byte of RAM, and an instructional demonstration of how key RAM locations in *Pitfall!* are used during play. We have made these examples available alongside s4LVE's source code.[2]

3.1 *Asteroids*

While the maximum amount of address space for ROM in the Atari 2600 was limited to 4 KiB, the amount of ROM in a game cartridge could be larger. Additional hardware in the game cartridge would enable the ROM's contents to be divided into memory banks, that the programmer would toggle between explicitly using instructions to access special memory addresses that acted as "soft switches." How frequently would a game flip memory banks? The bank switch demo (Figure 4) makes this visually apparent for *Asteroids* (an 8 KiB cartridge) by highlighting the currently active ROM bank in the spreadsheet ; this allows the game to be played

[2] https://github.com/ekaltman/javatari.js



	A	B	C	D	E	F	G	H	I	J	
0	Pitfall!										
1		0	1	2	3	4	5	6	7	8	9
2	008x	a0	c4	fe	f	80	2	41	0	ff	c
3	009x	18	0	13	4	0	f0	20	50	1	8
4	00Ax	7f	7f	7f	ff	ff	ff	ff	12	12	12
5	00Bx	7e	5e	76	3c	18	6e	fb	0	fd	21
6	00Cx	fe	b0	fd	60	fe	8	ff	68	ff	78
7	00Dx	ff	2	80	6f	0	12	34	56	19	50
8	00Ex	0	16	7c	88	5	0	0	0	0	20
9	00Fx	0	1f	0	0	0	0	0	0	0	0
10											

Figure 5: *Pitfall!*'s RAM usage visualized with s^{4}LVE (nonessential UI elements have been elided for clarity).

and the bank switching frequency to be easily observed simultaneously. The demo has uses for both conveying this information to an audience, as well as analysis: it quickly gives an intuitive sense as to how *Asteroids* is making use of memory.

3.2 The PCG of *Pitfall!*

In our second example, we explore how the game *Pitfall!* employs some clever procedural content generation (PCG) to fit 255 screens of jungle into a 4 KiB cartridge. The scheme hinges on the value—and interpretation—of a single byte in RAM, which is changed whenever the player flips from one screen of jungle to another [2]. The *Pitfall!* PCG demo (Figure 1) shows the way that the game interprets this one byte, dynamically updating as the jungle screen changes; the individual bits within the PCG byte are shown, and dynamic spreadsheet highlighting indicates how those bits translate into what is seen onscreen in the game.

3.3 *Pitfall!* in Motion

Our final example is meant to walk an interactor through *Pitfall!*'s use of some key RAM locations. The incremental, stepwise nature of this demo allows the player time to focus on each location as its use is revealed, and even see how their in-game activity affects the location. For example, when the X-coordinate of Pitfall Harry is revealed (Figure 5), the player can move back and forth to watch the memory location change. The selected locations are shown in the context of the Atari 2600's entire RAM contents continuously changing as the game is played, illustrating how s^{4}LVE scales with many updates on each time step.

4 FUTURE WORK AND CONCLUSIONS

Our immediate goal is to further extend the expressive power of both the DSL and the spreadsheet operations as we gain further experience developing concrete examples with s^{4}LVE. We may look to visualization-oriented languages like Protovis [3], Diderot [4], and Penrose [11], or the interface and study design of game-focused mixed-initiative UIs like Kwiri [7] and Cicero [6] for inspiration here.

s^{4}LVE is already a handy tool for understanding the behavior of memory locations over time, and its sophisticated state sequence analysis and flexible UI are significant improvements over the memory address exploration interfaces incorporated into game console emulators like BizHawk or FCEUX (which are largely oriented towards hacking, cheating at, or debugging games). That said, extending s^{4}LVE with other features of those interfaces—numerical inequalities over memory address contents, UI for searching and filtering addresses or sets of addresses, and so on—is a natural point of extension for the core language. Since we have the full power of regular expressions at our disposal, we could also make *capture groups* (both of byte sequences and of state sequences) available to the DSL and visualization context for more complex triggering conditions—or, potentially, to handle dynamic memory allocation.

s^{4}LVE's modular design requires a minimally invasive addition of hooks to the target emulator. This can be done by inserting two function calls into the emulator's inner loops. Therefore, it makes sense to extend the set of s^{4}LVE-supported emulators. Many such emulators can be cross-compiled to Javascript, maintaining easy shareability.

s^{4}LVE is a first step into a complex and deep application domain requiring a lot of specialized knowledge of reverse engineering, platform studies specifics, and experience making visualizations. We hope that we can make this branch of game studies a little less mysterious by continuing to improve s^{4}LVE's usability and by using it to illustrate the behavior of real games of interest to the game studies community. Of all our planned future work, this knowledge base of shareable, modifiable, and remixable "interactive documentation" is by far the most significant. We look forward to building this resource together with collaborators from all corners of game studies and every community of play.

ACKNOWLEDGMENTS

The third author's work is supported in part by the Natural Sciences and Engineering Council of Canada, grant RGPIN-2015-06359.

REFERENCES

[1] J. Aycock. 2015. Applied Computer History: Teaching Systems Topics through Retrogames. In *20th ACM Annual Conference on Innovation and Technology in Computer Science Education*. 105–110. https://doi.org/10.1145/2729094.2742583
[2] J. Aycock. 2016. *Retrogame Archeology: Exploring Old Computer Games*. Springer.
[3] Jeffrey Heer and Michael Bostock. 2010. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (2010), 1149–1156.
[4] Gordon Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John Reppy. 2016. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE transactions on visualization and computer graphics* 22, 1 (2016), 867–876.
[5] M. E. Lesk and E. Schmidt. 1975. Lex – A Lexical Analyzer Generator. Unix (7th edition) Programmer's Manual.
[6] T. Machado, D. Gopstein, A. Nealen, O. Nov, and J. Togelius. 2018. AI-Assisted Game Debugging with Cicero. In *2018 IEEE Congress on Evolutionary Computation*

(CEC). 1–8. https://doi.org/10.1109/CEC.2018.8477829

[7] Tiago Machado, Daniel Gopstein, Andy Nealen, and Julian Togelius. 2019. Kwiri - What, When, Where and Who: Everything you ever wanted to know about your game but didn't know how to ask. *CEUR Workshop Proceedings* 2313 (1 1 2019), 43–50.

[8] M. Mernik, J. Heering, and A. M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344.

[9] Joseph Carter Osborn, Ben Samuel, Michael Mateas, and Noah Wardrip-Fruin. 2015. Playspecs: Regular expressions for game play traces. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

[10] P. A. Peccin. 2015. Javatari – Online Atari 2600 Emulator. https://github.com/ppeccin/javatari.js

[11] Katherine Ye, Keenan Crane, Jonathan Aldrich, and Joshua Sunshine. 2017. Designing extensible, domain-specific languages for mathematical diagrams. In *Off the Beaten Track*.