Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Declarative Time-Based Animation using Domain Specific Language

Dr Moratanch N[1] | Srikaanth[2] | Priyadharsan[2] | Srinivasan[2]

[1]Undergraduate Student, Department of Computer Science and Engineering, SRC, SASTRA Deemed University, Kumbakonam, India

[2]Assistant Professor, Department of Computer Science and Engineering, SRC, SASTRA Deemed University, Kumbakonam, India

Correspondence
Corresponding to: Dr Moratanch N, Assistant Professor, Department of Computer Science and Engineering, SRC, SASTRA Deemed University, Kumbakonam,India
Email: authorone@gmail.com

Present address
This is sample for present address text this is sample for present address text.

Abstract

Animations are an essential element of the current interactive systems, where the visual behavior is supposed to change with time in a known and predictable fashion. Even with the ubiquitous presence of animation tools, time as a representable and analyzable entity is not easily captured especially where the animation behavior needs to be fine-grained. Most of the currently used methods of animation implicitly represent time by discrete keyframes or using percentages, thereby restricting the ability to infer time.

To overcome this drawback, it has been inspired by the GameScript, a domain-specific language that minimizes complexity with minimum programming in the creation of 2D games. GameScript represents every production-level object as a self-reliant entity, known as an Agent, and uses a Per- Agent System to control behavior through time. Based on this idea, the given approach proposes an approach, in which time is a first-class function, and time can be evaluated, serialized, and reasoned about in a systematic manner.

Within this context, time is evaluated as an unending operation and systematically translated into values to be implemented. At the time analysis step the expressions written in the DSL are interpreted and converted into representations that can be executed by general animation executor. Consequently, the definition of animation is divided into the two issues: temporal and visual realization.

The result of the synthesizing is eventually translated into Cascading Style Sheets (CSS) wherein keyframes and animation properties are automatically created. The system does not require any manual working with keyframes and percentage values by using a compiler-like workflow comprising of lexical, syntactic, and semantic analysis. The abstraction enables the expression of complex, time-constrained animations at a higher level, resulting in increased clarity, accuracy and maintainability and compatibility with existing web rendering pipelines.

KEYWORDS

Animation - Domain Specific Language (DSL); Animation Behaviour; Temporal Sampling; Keyframe Generation; Time-Aware Compilation; Declarative Animation Modeling

1 | INTRODUCTION

Animations are ubiquitous in contemporary digital media, appearing in applications ranging from video games to everyday visual displays such as electronic billboards. (Paper 1)As technological advancements have evolved, the process of scripting animations has become increasingly streamlined; however, challenges remain, particularly in achieving precise timing control and fine-tuning animation parameters. Current tools, including CSS animations and AI-driven solutions, often struggle with complex timing adjustments or granular control, especially when users require detailed customization.

Abbreviations: DSL, Domain Specific Language;

To address these challenges, we propose a simplified approach to creating CSS animations by integrating GameScripta domain-specific language (DSL)into the development workflow. (Paper 17)This integration facilitates the automatic generation of key CSS components, including @keyframes rules and related properties, thereby enhancing efficiency and accuracy in animation development.

Originally, GameScript was designed for creating interactive and visual content such as classic arcade-style games including Asteroid Destroyer and Tetris.(Paper 8) Its application to the web domain leverages its structured syntax, repeatability, and logical clarity. Importantly, we do not suggest that GameScript or our enhancements are intended to replace traditional CSS coding; rather, they serve as tools that generate CSS code efficiently. Our focus is on enabling smoother curve transitions, hover effects, focus states, active states, group interactions, property transformations, and peer-related logic.

While our approach offers significant advantages, it is important to acknowledge certain limitations inherent to the current architecture of GameScript, such as the implementation of grid-based functionalities.

We propose a methodology that converts a DSL built by us, characterized by a user-friendly, IF-ELSE structured syntax similar to natural language, into CSS animations through a series of processing stages. These include parsing, sampling, semantic interpretation, and a critical timing layer component. The timing layer plays a pivotal role in controlling the precise output of animations, enabling users to achieve desired effects with improved accuracy and ease.

2 | RELATED WORKS

This section looks at recent research done in game scripting, domain-specific languages, narrative control, machine assistance, and ease-of-use game development methodologies. Rather than separate or individual literature reviews, the discussion follows the continuity of major concepts in the literature as progression is made with respect to works that account for the weaknesses of earlier approaches and how these extensions contribute to the design of the proposed GameScript (GS) system.

2.1 | Rule-Based and Declarative Game Scripting

The early research in the domain of game scripting focused on reducing procedural complexity by representing game logics with rules and declarative constructs. Initial studies showed that from structured rule sets, game mechanics and levels could be generated automatically and that expressive gameplay does not necessarily require detailed procedural coding (Paper 1). This work established that rule-based representations would be feasible as a foundation for game logics. As rule-based systems attracted attention, researchers started to investigate their expressive power in practical settings. Investigations on declarative rule-based languages presented that compact formulations of rules can qualitatively record emergent behaviors and complex interactions within games (Paper 16). However, as these systems increase in size, concerns move to script organization, efficiency, and maintainability. Further investigation emphasized poorly designed or architected scripting languages that can hamper both development speed and runtime performance (Paper 5). Later approaches, to gain clarity and, particularly, long-term maintainability, introduced structured declarative models that explicitly separate game state from behavioral rules(Paper 18). Although this separation benefited readability and organization, most of the existing solutions tended to miss a unified and lightweight scripting approach that properly balances simplicity against expressive control. These observations lay the basis for GS, which uses a compact, rule-based scripting approach that is intended to remain readable even as game logic becomes more complex

2.2 | Domain-Specific Languages for Game Development

Based on the shortcomings of general-purpose scripting, there were efforts directed at domain-specific languages (DSLs) as a solution to increase the level of abstraction in game development. Some of the initial attempts in the field of DSLs were the incorporation of game-specific abstractions into host languages, allowing the description of behavior on multiple abstraction levels, yet using the existing execution environment (Paper 10). This was followed by further

research work in which extensible game description languages were developed to make changes in game rules and behavior programmable through these languages without requiring changes in game engines (Paper 13). Although this made game development more flexible, this area of research again emphasized the need for well-defined domain concepts to prevent over-engineering. With further advances in DSL research, more emphasis was given to serious games like educational and games for educational institutions. DSL for modeling adventure and educational games successfully applied domain-specific concepts to define gameplay (Paper 14). Further evolution to the language-driven development demonstrated that the definition of game logic before the engine construction allows achieving better modularity and reusability of the code (Paper 15). However, most DSLs still have steep learning curves or rigid syntax. GS addresses this gap by offering a minimal, rule-oriented DSL that maintains abstraction advantages and stays comprehensible and easy to modify.

2.3 | Game Scripting-Engines and Engine Architecture

As high-level scripting and DSLs became commonplace, interest in research output started to shift toward the way such scripts are executed with efficiency within game engines. Performance and memory limits were early points of interest where lightweight scripting engines, fit for resource-constrained platforms like mobile, were proposed(Paper 2). Such an engine proved that a reduced execution model can drastically minimize the footprint of the resources. Parallel to this, other research focused on more general aspects of the architecture of the engines, focusing on modular design and performance optimization to handle the continuous growth in game logic complexity (Paper 9). Such architectures did improve scalability but at the same time increased system complexity. The long-term maintainability concerns were indeed confirmed in later studies that focused on code quality and sustainability aspects of some of the most popular open-source game engines (Paper 11).In response to these challenges, object-oriented design patterns were used to increase the reusability and flexibility of the game (Paper 28). Yet, this approach tends to move the complexity altogether, making the design understandable only by expert designers. These kinds of advancements imply that simplifying the scripting layer itself, instead of using complex engine architecture, would be more long-term, which GS aims to address.

2.4 | Narrative, Dialogue, and Interactive Story Scripting

Game scripting development also incorporates narrative-oriented and interactive story solutions. Structured game scripts have been found early on in research to facilitate branching scenarios and conversations, making it feasible for any individual to develop interactive storylines without necessarily being programmatically competent (Paper 4). These hypotheses were later verified by game implementations where the scripts handled conversations, puzzle-solving, and storylines in adventure-type games (Paper 7). With the rising interest in Narrative Automation, researchers did look into AI solutions. Some recent work used Large Language Models to automatically create an interactive story sequence (Paper 21). But this work posed its own problems regarding predictability and execution. Some related work examined properties in game dialogues (Paper 22), while others explored ways to model behavioral data to handle non-linear narratives (Paper 25). Although these approaches make progress for narrative flexibility, they frequently do so at the expense of transparency and simplicity. By contrast, GS focuses on explicit, rule-based scripting of narratives, giving designers more control over narrative logic without resorting to complex AI-intensive systems.

2.5 | Educational Games and Gamification Systems

The requirement for simple scripting tools is particularly important in education-based as well as gamification-based systems. The initial research proved that knowledge modeling through scripting can be useful in structuring game-based learning activities in education-based games (Paper 3). Simplified scripting engines with a reduced instruction set were later developed to facilitate scripting activities even for beginners, allowing students to conduct scripting activities in games without having in-depth knowledge in programming (Paper 6). On top of this, the development of authoring systems followed to enable the creation of gamified learning activities by high-level scripting (Paper 8). Research continued with collaborative learning, team evaluation, and games for inquiry-based learning (Papers 24, 26,

27, and 30). Although the emphasis of these studies now lies on pedagogical outcome rather than technical design, the significance of understandable representations of game logic remains, thus upholding the importance of GS.

2.6 | Model-Driven and Visual Game Design Methods

Alongside script-based solutions, model-based and graphical abstractions have been investigated to further minimize development time. Domain-specific modeling languages and model-driven development allowed showing the capability to transform high-level models directly into game executables with less reliance on programming (Papers 17 19). Graph DSLs furthered this concept to enable designers to represent game concepts in graphical forms (Paper 20). Even as they provide strong abstraction over low-level details, they can be rather restrictive in terms of expressivity. Such a trade-off between abstraction and expressibility serves to emphasize GS as a middle path that combines scripting expressibility with simplicity.

2.7 | Game Design and Conceptual Planning

Some research concentrates on supporting conceptual and nascent designs as opposed to executable logic. Game sketching assists in quickly evaluating designs (Paper 23), and game design documents aid in maintaining consistency and theme integrity (Paper 29). It can be seen that these contributions are useful for the purpose of planning, but they do not cover the specification and implementation of game behavior. Lastly, research on the use of blockchain and game theory for federated learning incentivization mechanisms (Paper 12) can be considered irrelevant to this project since it neither relates to game scripting nor to methodologies on developing games.

SUMMARY OF RELATED WORK

- Previous work involved rule-based and declarative scripting to alleviate procedural complexity while preserving expressive game behavior.
- Domain-specific languages enhanced abstraction in game development but tended to impose inflexible syntax or learning overhead.
- Game scripting engines concentrated on efficiency and modularity, sometimes adding architectural complexity.
- Narrative and educational scripting paradigms prioritized usability but compromised transparency and predictability.
- Model-driven and visual approaches made design simpler but restrictive in expression, thus underlining the need for a lightweight scripting solution.

3 | PROPOSED METHODOLOGY

We introduce a conceptual framework that will make the creation of animations easier and allow performing more granular control over animation parameters. The methodology entails the consumption of the temporal information as one of the essential input parameters in a domain-specific language (DSL) syntax that comprises the full descriptive semantics of animation sequence desired. This DSL input is then manipulated and converted through a methodical conversion process to machine readable and understandable numerical values in accordance with CSS property values. The transformed values are then systematically injected into the CSS rendering pipeline, thus, producing the animation with built-in time control, which in turn is able to provide the ability to time, sequence and synchronise control mechanisms that are part of the animation cycle.

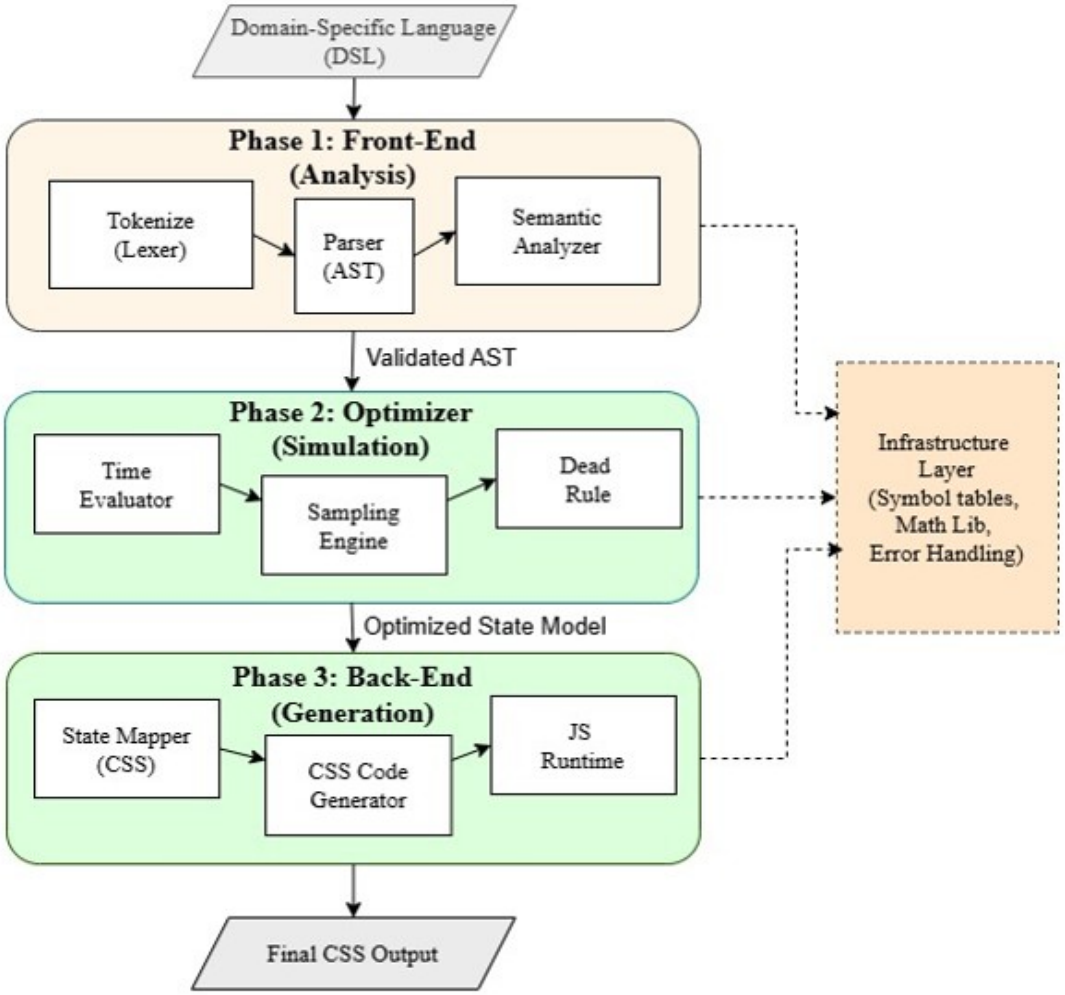

Figure 1: Architecture Diagram of the GameScript System

FIGURE 1 Architecture diagram of the GameScript compilation pipeline

3.1 | Temporal Analysis

The Domain-Specific Language (DSL) requires the end-user to define a temporal parameter that establishes the specific timelines in which the action should be executed, and the specific set of operations that should be instantiated at such timelines. It shares the underlying computational logic of a conditional (if-then) construct whereby the active modulation of the animation parameters is the predicated parameter, that is, the parameter of opacity, based on continuous and periodic functions represented as trigonometric sine and cosine functions. The choice of these functions is based on their intrinsic simplicity, periodicity and determinism that, together, allow the production of repeatable and predictable visual effects needed to produce consistent animation effects in graphical user interfaces.

Opacity, represented as a time-varying constant, O(t) is represented as a continuous function of time t, and is gradually increasing as the animation advances through the animation phase. All these functions make transitions between visual levels of transparency smooth, and they are the workings of developing the animated element within the temporal domain. Other functions like tangent are not included on purpose because of their predisposition to asymptotic discontinuities and unstable behavior that cannot be trusted to maintain animation fidelity. Even though linear interpolation models may be used, it is possible that the interpolation may create sharp or abrupt transitions hence compromising visual smoothness.

Although the sine and cosine functions form the main modeling technique, more complicated parametric curves, including Bezier splines, can be combined in the case they are needed to provide more detailed or stylistically rich animation effects. The temporal profile T (t) obtained through a sine-based calculation provides a parameterizable temporal profile that can be directly inserted into Cascading Style Sheets (CSS) animation keyframes and transition definitions, compatible with web-based rendering engines and making it easy to control the transitions between animated changes in opacity.

3.2 | Implementation Computational Toolkit

The Domain-Specific Language (DSL) requires the end-user to define a temporal parameter that establishes the specific timelines in which the action should be executed, and the specific set of operations that should be instantiated at such timelines. It shares the underlying computational logic of a conditional (if-then) construct whereby the active modulation of the animation parameters is the predicated parameter, that is, the parameter of opacity, based on continuous and periodic functions represented as trigonometric sine and cosine functions. The choice of these functions is based on their intrinsic simplicity, periodicity and determinism that, together, allow the production of repeatable and predictable visual effects needed to produce consistent animation effects in graphical user interfaces.

Opacity, represented as a time-varying constant, O(t) is represented as a continuous function of time t, and is gradually increasing as the animation advances through the animation phase. All these functions make transitions between visual levels of transparency smooth, and they are the workings of developing the animated element within the temporal domain. Other functions like tangent are not included on purpose because of their predisposition to asymptotic discontinuities and unstable behavior that cannot be trusted to maintain animation fidelity. Even though linear interpolation models may be used, it is possible that the interpolation may create sharp or abrupt transitions hence compromising visual smoothness.

Although the sine and cosine functions form the main modeling technique, more complicated parametric curves, including Bezier splines, can be combined in the case they are needed to provide more detailed or stylistically rich animation effects. The temporal profile T (t) obtained through a sine-based calculation provides a parameterizable temporal profile that can be directly inserted into Cascading Style Sheets (CSS) animation keyframes and transition definitions, compatible with web-based rendering engines and making it easy to control the transitions between animated changes in opacity.

4 | PROCESSING PIPELINE

The architecture of the proposed system is described in this section. It is designed in a classical compiler pipeline framework, except that it is made to accept time-based animation specification in the form of a domain-specific

language. The system takes a DSL input and analyzes it at the front-end stage of analysis, optimizes it with time-awareness, and produces animal code.

The architecture is divided into three key steps, namely, the front-end processing, time evaluation optimization, and code generation in the back-end. The stages are tasked with the responsibility of gradually converting the input specification into a form that can be executed.

4.1 | Domain-Specific Language

A domain-specific language is used as the input of the proposed system. The DSL is intended to describe animation behavior through time in an explicit manner, which is hard to describe using the standard constructs of CSS animation. The language permits the user to define time intervals, define state transitions in between time intervals, and states the connections among various animation elements, e.g. sequentialallel execution. The DSL is declarative and thus the system can reason about animation behavior without having to make reference to the details at the implementation level.

4.2 | DSL Program Analysis Pipeline

The input is processed by lexical analysis, which receives the DSL input as a stream of characters and generates it into a stream of tokens. These tokens include time literals, identifiers, keywords, and operators defined by the language specification. This step ensures that the input conforms to the lexical rules of the DSL and identifies errors such as invalid literals or unsupported symbols. The output of this phase is a validated token stream that serves as input to the parser.

Syntactic analysis is then performed to verify adherence to the grammar of the DSL using the generated token stream. During this phase, the structural relationships among animation definitions and time-scoped blocks are determined. The parser constructs an abstract syntax tree (AST) that represents the hierarchical structure of the input program. Syntactic errors, including missing elements, malformed constructs, or invalid nesting, are detected and reported at this stage.

Semantic analysis validates the logical correctness of the parsed structure. This includes verification of time intervals, consistency of animation properties, and reference validation of animation elements. Conflicts, such as incompatible transformations applied to the same property within overlapping time ranges, are identified during the semantic phase. The output of this stage is an annotated representation that captures both the structural and temporal semantics of the DSL program.

4.3 | Time Evaluation and Optimization

Following the semantic validation phase, the program advances to the time evaluation and optimization stage, which explicitly interprets temporal aspects and prepares the intermediate representation for efficient code generation. The time evaluation component normalizes and scales individual time values and constructs a unified timeline encompassing all animation components. Declarative time specifications are translated into evaluable representations to ensure consistent interpretation across the system.

Subsequently, the sampling engine discretizes continuous time-based behavior by estimating animation states at fixed temporal intervals. This process generates discrete state representations suitable for CSS keyframes while balancing temporal accuracy with operational feasibility. Sampling ensures that the resulting animation maintains visual smoothness without producing excessive keyframes.

Finally, optimization rules are applied to reduce redundancy and enhance execution performance. These rules combine compatible animation segments, eliminate unnecessary keyframes, and resolve temporal overlaps according to predefined precedence constraints. Throughout this phase, the semantic integrity of the DSL is preserved while improving execution efficiency and preparing the animation model for reliable code generation.

4.4 | Code Generation

Following the optimization phase, the back-end converts the optimized intermediate representation into executable code suitable for deployment within standard web rendering environments. During the state mapping stage, abstract animation states produced by the time evaluation and optimization phases are translated into concrete states compatible with CSS. This translation process converts high-level animation intent into precise CSS properties and well-defined state transitions according to the analyzed and normalized time scale.

During CSS generation, the system automatically produces CSS code that includes keyframe specifications and animation declarations. These outputs conform to standard CSS animation specifications and accurately reflect the temporal behavior defined in the DSL. Keyframe percentages, property values, and easing functions are derived directly from the evaluated timeline, eliminating the need for manual keyframe construction or percentage-based calculations by the developer.

Finally, lightweight runtime support is employed where dynamic triggering, coordination, or synchronization of animations is required. This runtime layer complements the generated CSS by handling event-based activation and sequencing without duplicating CSS functionality. As a result, the system achieves efficient execution while preserving compatibility with existing web-based animation pipelines.

SUMMARY

- The system follows a compiler-inspired pipeline that processes the DSL input through lexical, syntactic, and semantic analysis, producing an annotated representation of animations.
- Time evaluation and sampling scale and discretize animation timelines, translating continuous behavior into discrete states suitable for CSS keyframes while maintaining temporal accuracy.
- Optimization rules reduce redundancy, resolve temporal overlaps, and combine compatible animation segments, enhancing performance while preserving the semantics of the DSL.
- The code generation and runtime support convert the optimized representation into CSS-compatible states, keyframes, and dynamic animation management, ensuring efficient, accurate, and maintainable execution.

5 | IMPLEMENTATION

The implementation of the proposed system is centered around a set of core algorithms that operate on the validated intermediate representation produced by the DSL analysis pipeline. These algorithms handle temporal ordering, timeline construction with conflict resolution, and CSS keyframe generation.

5.1 | Implemented Algorithms

The following algorithms describe the core implementation stages of the proposed system.

Algorithm 1 Temporal Event Ordering

Require: Set of animation events $E = \{e_1, e_2, \ldots, e_n\}$ with associated time values
Ensure: Chronologically ordered event list E'
for all $e_i \in E$ **do**
  Extract timestamp t_i from e_i
end for
Sort E in ascending order of t_i using a stable sorting strategy
$E' \leftarrow E$

Result:
Deterministic temporal ordering is achieved regardless of the declaration order in the DSL. Events with identical timestamps preserve their relative order, ensuring predictable execution for both sequential and parallel animations. **return** E'

Algorithm 2 Timeline Construction and Conflict Resolution

Require: Temporally ordered event list E'
Ensure: Conflict-free property timelines $\{T_p\}$
for all animation properties p **do**
  Initialize an empty timeline T_p
end for
for all event $e \in E'$ **do**
  Convert e into a time interval $[t_{start}, t_{end}]$
  Insert the interval into the corresponding T_p
  if overlapping intervals are detected in T_p **then**
    Resolve conflicts using predefined precedence rules
    Merge or trim intervals accordingly
  end if
end for

Result:
Overlapping transformations applied to the same property are resolved deterministically, ensuring that a single valid transformation is active at any given time instant and preventing unstable animation states. **return** $\{T_p\}$

Algorithm 3 CSS Keyframe Generation

Require: Optimized timelines $\{T_p\}$ and total animation duration D
Ensure: CSS keyframes and corresponding animation declarations
for all timeline T_p **do**
  for all state change occurring at time t in T_p **do**
    Normalize time: $k \leftarrow (t/D) \times 100$
    Generate a keyframe at $k\%$ with the corresponding property value
  end for
  Eliminate redundant keyframes
end for
Emit CSS @keyframes definitions and animation rules

Result:
Compact and valid CSS animations are generated automatically. Redundant keyframes are removed while preserving visual smoothness and accurately reflecting the temporal semantics of the DSL. **return** Generated CSS code

6 | DATASETS

As the proposed approach does not rely on an existing deterministic benchmark dataset, evaluation is conducted using a curated set of representative animation tasks. These tasks model common time-based user interface animation behaviors and are used as structured inputs at the specification level for validating the proposed domain-specific language. The dataset is designed to capture fundamental animation patterns involving temporal progression, easing, concurrency, and repetition.

Fade-in over time. The element remains initially invisible and transitions to a fully visible state over a fixed duration. The initial opacity is set to 0 and progressively increases to 1 over a duration of 1 second.

```
over 1s:
  opacity 0 -> 1
```

Horizontal translation with easing. An element moves horizontally along the X-axis using a smooth ease-in-out timing function. The animation starts at position $x = 0$ and translates to $x = 100$ pixels over a duration of 2 seconds.

```
over 2s ease-in-out:
  translateX 0 -> 100
```

Scale with opacity. An element simultaneously scales and changes opacity. The scale increases from 0.8 to 1.0 while the opacity transitions from 0 to 1 over a duration of 1.5 seconds.

```
over 1.5s:
  scale 0.8 -> 1
  opacity 0 -> 1
```

Staggered animation. Multiple elements animate sequentially with a fixed temporal offset between their start times. Each element animates over 1 second, with a stagger delay of 0.2 seconds, using vertical translation and opacity effects.

```
stagger 0.2s:
  over 1s:
    opacity 0 -> 1
    translateY 20 -> 0
```

Looping animation with delay. An animation is repeated indefinitely with a fixed delay between successive iterations. Each iteration performs a full rotation from 0 to 360 degrees over 1 second, followed by a delay of 0.5 seconds.

```
loop infinite:
  over 1s:
    rotate 0 -> 360
  delay 0.5s
```

These representative tasks collectively evaluate the expressiveness and correctness of the proposed DSL across single-property animations, multi-property synchronization, easing behavior, staggered execution, and looping constructs. This task-based dataset enables systematic validation of temporal reasoning, conflict resolution, and CSS code generation without requiring external animation benchmarks.

REFERENCES

1. Li J, Wang Q, Jayasinghe D, Park J, Zhu T, Pu C. Performance overhead among three hypervisors: an experimental study using Hadoop benchmarks. *IEEE International Congress on Big Data*. 2013;9–16.
2. Soriga SG, Barbulescu M. A comparison of the performance and scalability of Xen and KVM hypervisors. *RoEduNet*. 2013;1–6.
3. Hwang J, Zeng S, Wu FY, Wood T. A component-based performance comparison of four hypervisors. *IFIP/IM*. 2013;269–276.
4. Che J, He Q, Gao Q, Huang D. Performance measuring and comparing of virtual machine monitors. *EUC*. 2008;381–386.
5. Shirinbab S, Lundberg L, Ilie D. Performance comparison of KVM, VMware and XenServer using a large telecommunication application. In: *Proceedings of the 5th International Conference on Cloud Computing, GRIDs, and Virtualization*; 2014:114–122.
6. Yu L, Chen L, Cai Z, Shen H, Liang Y, Pan Y. Stochastic load balancing for virtual resource management in datacenters. *IEEE Transactions on Cloud Computing*. 2016;4:1–14.
7. Adufu T, Choi J, Kim Y. Is container-based technology a winner for high performance scientific applications? *IEICE*. 2015;507–510.
8. Soltesz S, Pötzl H, Fiuczynski M, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *EuroSys*. 2007;275–287.
9. Dua R, Raja AR, Kakadia D. Virtualization vs containerization to support PaaS. *IC2E*. 2014;610–614.
10. Adufu T, Choi J, Kim Y. Is container-based technology a winner for high performance scientific applications? *APNOMS*. 2015;507–510.
11. LXC. Linux Containers. 2016. Available at: https://linuxcontainers.org/
12. Pahl C. Containerization and the PaaS cloud. *IEEE Cloud Computing*. 2015;2(3):24–31.
13. Wu R, Deng A, Chen Y, Blasch E, Liu B. Cloud technology applications for area surveillance. *NAECON*. 2015;89–94.
14. OpenVZ. OpenVZ Virtuozzo Containers. 2016. Available at: https://openvz.org/
15. Ismail BI, Mostajeran Goortani E, Bazli Ab Karim M, et al. Evaluation of Docker as Edge computing platform. *ICOS*. 2015;130–135.
16. Docker. Docker. 2016. Available at: https://www.docker.com/
17. Anderson C. Docker [software engineering]. *IEEE Software*. 2015;32(3):102–c3.
18. Kan C. DoCloud: an elastic cloud platform for web applications based on Docker. *ICACT*. 2016;478–483.
19. Preeth EN, Mulerickal FJP, Paul B, Sastri Y. Evaluation of Docker containers based on hardware utilization. *ICCC*. 2015;697–700.
20. Gkortzis A, Rizou S, Spinellis D. An empirical analysis of vulnerabilities in virtualization technologies. In: *Cloud Computing Technology and Science Conference*; 2017:533–538.
21. Okur MC, Buyukkececi M. Big data challenges in information engineering curriculum. *EAEEIE*. 2014;1–4.
22. Jain R, Iyengar S, Arora A. Overview of popular graph databases. *ICCCNT*. 2013;1–6.
23. Cassandra. Apache Cassandra. 2016. Available at: http://cassandra.apache.org/
24. Carpenter J, Hewitt E. *Cassandra: The Definitive Guide*. O'Reilly Media; 2016.
25. Lourenco JR, Abramova V, Cabral B, Bernardino J, Carreiro P, Vieira M. NoSQL in practice: a write-heavy enterprise application. *IEEE International Congress on Big Data*. 2015;584–591.
26. Chebotko A, Kashlev A, Lu S. A big data modeling methodology for Apache Cassandra. *IEEE International Congress on Big Data*. 2015;238–245.
27. Niyizamwiyitira C, Lundberg L. Performance evaluation of SQL and NoSQL database management systems in a cluster. *International Journal of Database Management Systems*. 2017;9(6):1–24.
28. Klein J, Gorton I, Ernst N, Donohoe P, Pham K, Matser C. Performance evaluation of NoSQL databases: a case study. *PABS'15*. 2015:5. doi:https://doi.org/10.1145/2694730.2694731.
29. Bakkum P, Skadron K. Accelerating SQL database operations on a GPU with CUDA. *GPGPU-3*. 2010;94–103.
30. Breß S, Heimel M, Siegmund N, Bellatreche L, Saake G. GPU-accelerated database systems: survey and open challenges. In: Hameurlain A, Küng J, Wagner R, et al., eds. *Transactions on Large-Scale Data- and*

Knowledge-Centered Systems XV. Lecture Notes in Computer Science, Vol. 8920. Berlin, Germany: Springer; 2014:1–35.