A Declarative Model for Entity-Component-System Game Descriptions

Breno Guimarães[1], **Geraldo Xexéo**[1]

[1]LUDES - Laboratório de Ludologia, Engenharia e Simulação
Programa de Engenharia de Sistemas e Computação – COPPE
Universidade Federal do Rio de Janeiro

{breno, xexeo}@cos.ufrj.br

Abstract. Introduction: *The Entity-Component-System (ECS) architectural pattern has gained wide adoption in game engines for its performance and modularity benefits, yet game-specific logic remains expressed in code even in ECS-based engines, hindering portability and formal analysis of game descriptions.* ***Objective:*** *We investigate whether a complete, executable game can be described as pure structured data by augmenting the ECS model with a declarative reactive rule mechanism, and characterize the resulting model's computational and semantic properties.* ***Methodology:*** *We define the DECS (Declarative Entity-Component-System) model through a structured runtime state and small-step operational semantics, specifying a three-phase event-driven execution loop. We characterize the Behavior component as a distributed production system, conjectured via an informal correspondence argument to have expressive power equivalent to register machines, and define a conformance criterion that decouples execution semantics from engine capabilities.* ***Results:*** *A reference DECS-conformant engine successfully executes ten complete game descriptions spanning genres from physics puzzles to visual novels without additional game-specific imperative code beyond its declared capability set, providing evidence that the proposed specification is realizable. In addition, the conformance criterion makes executability against any given engine decidable at load time from its declared capability set.*
Keywords *Entity-Component-System, Declarative Programming, Data-Driven Game Development, Game Engines.*

1. Introduction

Modern video games have grown substantially in scale and complexity, making software architecture choices increasingly critical to their development [Politowski et al. 2021]. In this scenario, the Entity-Component-System (ECS) pattern has emerged from the industry as a leading architectural approach that organizes programs around data rather than object hierarchies, enabling high performance, modularity and scalability through better CPU cache efficiency, parallelism, and runtime flexibility [Bayliss 2022] [Voisard et al. 2025].

Established engines such as Unity and Unreal Engine have incorporated ECS as an optional high-performance layer (DOTS and Mass, respectively), while newer projects such as Bevy and Flecs adopt ECS as their exclusive architectural foundation. Despite the data-orientation of ECS, game logic in all mentioned engines remains in arbitrary, engine-specific programs, creating a barrier for non-programmers, hindering portability, and making formal analysis of game descriptions impractical.

These limitations motivate the investigation of an alternative approach. We take as our founding hypothesis that, if an engine provides systems covering a game's mechanical domains (physics, collision, movement, rendering, input) and the component model is extended with a reactive rule mechanism expressing arbitrary game-specific behavior, then a complete, playable game can be described entirely as declarative data. This does not eliminate computation, but replaces imperative game-specific code with a declarative specification, relocating the programming boundary from game designers to engine developers.

We address this idea through **DECS**, a declarative model with explicitly defined operational semantics, validated through a reference conformant engine and ten complete and distinct game descriptions.

2. Background and Related Work

The Entity-Component-System pattern structures real-time applications around three concepts: an **Entity** (a unique identifier), a **Component** (a typed data container attached to an entity), and a **System** (a logic unit operating over all entities with a required component set) [Voisard et al. 2025].

This pattern leads to Data-driven development, which externalizes game content to structured data files, separating configurable state from executable logic. [Fischbach et al. 2017] argued that state management concerns can be partially externalized in ECS-based systems, a direction DECS extends to full game logic. [Wiebusch e Latoschik 2015] proposed semantic trait decoupling of ECS components from systems, improving reusability. [da Silva et al. 2021] implemented a scripting system for game development on top of an ECS framework, illustrating that code-based logic remains the dominant approach even in data-oriented architectures. DECS addresses this by relocating game-specific logic from imperative code into declarative reactive rules, provided the engine's system repertoire already covers the game's mechanical domains.

[Raffaillac e Huot 2019] built Polyphony, an ECS-based reactive GUI toolkit demonstrating that declarative reactive rule mechanisms can be embedded within the component model. DECS scales this pattern from GUI interaction to full game development. [Marín-Lora et al. 2020] takes a related direction, proposing game engines whose object behaviour is defined through declarative condition/action rules without scripting.

Game description languages focused on board games are a parallel research track that follows the idea that the game description is independent of game execution. Most of them originated from the Stanford Game Description Language (GDL) [Love et al. 2006], which formalizes gameplay rules via Datalog-like logic, targeting finite turn-based games and specifying legal state transitions for AI reasoning rather than behavioral event reactions for real-time engines. GDL only covers complete information games; more powerful proposals extend it along several axes, including incomplete information, introspection, and richer game-oriented component syntax [Thielscher 2010, Thielscher 2017, Piette et al. 2020]. Unlike DECS, GDL leaves the execution model to an external interpreter and does not treat behavior as a first-class concern.

For video games, the main lineage begins with the Video Game Description Language (VGDL) agenda [Ebner et al. 2013]. Schaul's PyVGDL made this agenda

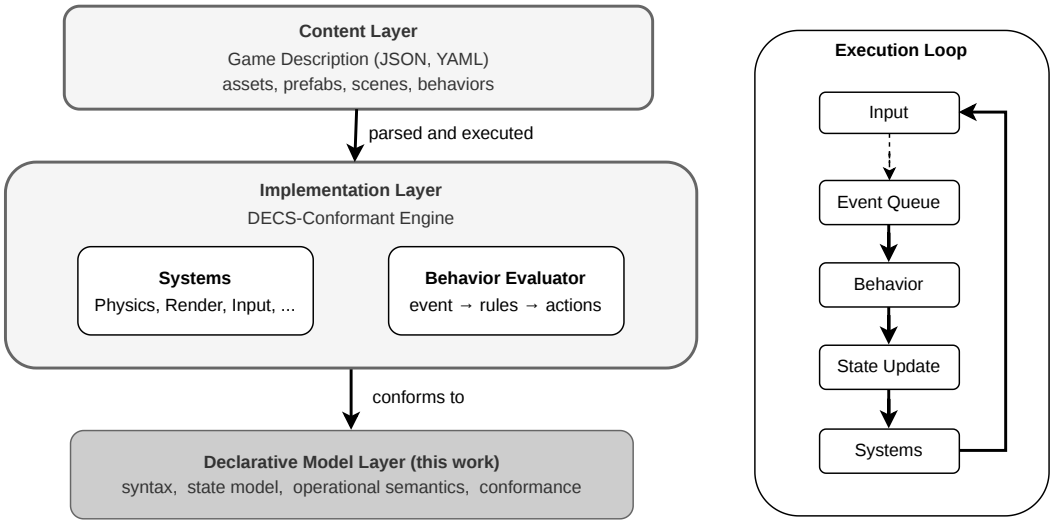

Figure 1. DECS three-layer architecture and execution loop. The model layer (this work) defines syntax, state model, and operational semantics. The implementation layer (conformant engines) executes game descriptions on any platform. The content layer (game descriptions) holds concrete game data in formats such as JSON or YAML.

executable as a high-level language for 2D video games, separating level descriptions from game dynamics and organizing descriptions around level mappings, sprite sets, interaction rules, and termination conditions supported by an extensible ontology of object behavior, physics, movement, spawning, transformation, and collision effects [Schaul 2013]. This model became central to the General Video Game AI framework, where VGDL is used to define benchmark games for agents, game generation, and level generation [Pérez-Liébana et al. 2019], and was later ported to Unity through UnityVGDL [Johansen et al. 2019]. Other video game DSLs address grid-based puzzle games [Julia e van Rozen 2023]; generative interactive systems and experimental game mechanics as proof search [Martens 2015]; and multi-agent systems whose actors have properties and predicate-logic behavior rules [Marín-Lora et al. 2020].

These approaches demonstrate that executable video game descriptions can be declarative, analyzable, and useful for design or AI research. However, while prior work provides languages for specific game classes or AI benchmarking, DECS provides formal operational semantics, a first-class reactive rule mechanism embedded in the component model, and a conformance criterion that decouples the specification from any particular engine.

3. The DECS Model

DECS is defined as a **declarative model** of the structure and execution semantics of game descriptions, independent of implementation. Figure 1 illustrates this architecture. It distinguishes three layers.

3.1. Game Description Structure

A DECS game description G is a 5-tuple: $G = (T, S, I, \Gamma, s_0)$ where $T : Id \rightarrow ComponentMap$ is the entity template set, a finite map giving each named template

its initial component data; $S : Id \rightarrow SceneData$ is the scene definition set, a finite map where each scene specifies an initial entity population referencing T, plus layout metadata; I is the abstract input set, mapping abstract input identifiers to hardware event patterns; $\Gamma : ContainerKey \rightarrow ContainerData$ is the global persistence store, a finite map from named keys to system-owned data persisting across scene transitions; and $s_0 \in \mathrm{dom}(S)$ is the initial scene identifier. Systems that require persistent global data declare named containers at the top level of the game description (for example, `attributes` for game variables, `localization` for string tables, and `dialogues` for dialogue trees), from which Γ is initialised. No container key has special status, all are system-defined.

A BNF Grammar of G is given in Section 4.

3.2. Runtime State Model

The runtime state σ of a DECS execution is: $\sigma = (E,\, M,\, Q,\, \Gamma,\, \varphi)$ where E is a finite *ordered* set of active entity instances; $M : E \times ComponentName \rightarrow ComponentData$ is the component mapping; Q is a FIFO-ordered event sequence, each event a tuple $(type,\, target\_spec,\, payload)$; $\Gamma : ContainerKey \rightarrow ContainerData$ is the global persistence store, persisting across scene transitions; and $\varphi \in \mathrm{dom}(S)$ is the active scene.

Entity instantiation. When entity template $p \in \mathrm{dom}(T)$ is instantiated with override map θ, a fresh e_{new} is appended to E, with $M(e_{new}, \cdot)$ initialized from $T(p)$'s component map with θ applied on top.

3.3. Operational Semantics: The Game Loop

The game loop is a state transition relation $\delta : \sigma \rightarrow \sigma'$. One iteration proceeds in three ordered steps:

1. **Input Processing:** Hardware events are translated via I into abstract input events and enqueued to Q.
2. **System Updates:** Systems are abstract transition functions over σ, whose composition is constrained by a dependency relation forming a directed acyclic graph. Step 2 follows any topological ordering of this graph, so systems with no declared dependency between them *may execute concurrently*.
3. **Behavior Evaluation:** Events are dequeued from Q in FIFO order. The formal small-step inference rules are:

[Rule-Fire]

$$\frac{\mathrm{eval}(c_i,\, e,\, ev,\, \sigma) = true \quad \forall\, c_i \in \mathrm{conditions}(r)}{\langle \sigma,\, e,\, r,\, ev \rangle \;\longrightarrow\; \mathrm{exec}(\mathrm{actions}(r),\, e,\, ev,\, \sigma)}$$

[Rule-Eval] For each event $ev = (type,\, target\_spec,\, payload)$, let $T = \mathrm{resolve\_target}(ev.target\_spec,\, e,\, ev,\, \sigma)$. For every $e \in T$ in the total order of E, and for every rule r in $M(e, Behaviors)[type]$ in list order, Rule-Fire is applied when its condition holds. Multiple rules for the same event fire sequentially (non-exclusive). Conditions within a single rule are pure read operations on σ (they never modify state), so a conformant engine may evaluate them concurrently without affecting correctness. Action execution remains strictly sequential within each rule.

3.4. Condition Evaluation and Action Execution

Source resolution. The source field src of a condition is resolved by $\mathrm{resolve\_src}$ to a data record d from one of four abstract categories: the triggering entity's own component data $M(e, C)$; the component data of a partner entity referenced in the event payload, if present; the event payload itself; or the global persistence store Γ. A conformant engine maps its source identifiers to these four categories.

Target resolution. The target field of an action or event is resolved by $\mathrm{resolve\_target}$ to a subset of active entities:

$$\mathrm{resolve\_target}(t, e, ev, \sigma) \subseteq E$$

The specifier t is instantiated from two distinct sources depending on the call site: during Rule-Eval, $t = ev.target\_spec$, determining which entities evaluate the rule; during action execution, t is the action's own `target` field (Table 1), which may differ from $ev.target\_spec$ — for instance, a rule may evaluate under `broadcast` while one of its actions targets only `self`. Three categories of specifier are required by the model: one resolving to the triggering entity alone; one resolving to the partner entity referenced in the event payload, if present; and one resolving to all active entities. Conformant engines may define additional specifiers as engine-level capabilities.

Condition evaluation. For condition $c = (src, C, P)$:

$$\mathrm{eval}(c, e, ev, \sigma) \;=\; \text{let } d = \mathrm{resolve\_src}(src, C, e, ev, \sigma) \text{ in } \forall (k, p) \in P : \mathrm{check}(p,\, d.k)$$

Action execution. Actions are partial state transition functions applied sequentially within each rule. Table 1 presents representative semantics only; the remaining grammar-level actions (`add`, `multiply`, `random`) follow the same state-transition style, differing only in the value computation applied to the target field.

Table 1. Representative action semantics (not exhaustive). T denotes the resolved target set; D a finite value domain.

Action	State transition
$\texttt{set}(T, C, k, v)$	$\forall e' \in T :\ M'(e', C).k := v$
$\texttt{spawn}(p, \theta)$	$E' := E \cup \{e_{new}\}$; M' init. from $T(p)$ with θ
$\texttt{destroy}(T)$	$E' := E \setminus T;\ \mathrm{dom}(M') := \mathrm{dom}(M) \setminus T$
$\texttt{emit}(ev, T)$	$Q' := Q +\!\!+ [(ev, T)]$
$\texttt{load\_scene}(s)$	$\varphi' := s;\ E' := \emptyset;\ M'$ init. from $S(s);\ \Gamma' := \Gamma$

3.5. The Behavior Component

The Behavior component in DECS is a *distributed production system* [Forgy 1982, Brownston et al. 1985], a rule-based formalism of the form *condition* $\rightarrow$ *action* in which each entity instance maintains its own rule base rather than sharing a single global working memory, and rule firing is driven by an event queue. The key structural difference from classical production systems is scope: DECS scopes rule evaluation to individual entity instances rather than a global flat memory, connecting DECS to theory with known computational properties (including Rete-style indexing for efficient condition

matching [Forgy 1982]) that can inform future engine implementations. This per-entity scoping is the property that enables code-free game descriptions, since reactive logic can be fully specified in data without a global controller.

Conjecture. The Behavior component gives DECS expressive power equivalent to register machines [Minsky 1967], modulo finite memory bounds. We do not provide a mechanized proof in this paper; we outline the informal correspondence and leave formal verification to future work. Component data fields act as addressable registers, readable and writable by conditions and actions respectively; conditions implement conditional branching (a rule fires only if its predicates hold); and the `emit` action, by re-enqueueing events that a rule base can react to, provides the unbounded control flow needed for iteration and looping. These correspond to the classical sufficient ingredients used in register-machine constructions [Minsky 1967], but we have not yet constructed the explicit encoding required to establish it rigorously.

4. Conformance and Reference Implementation

4.1. DECS Conformance Criterion

Definition (DECS-Conformant Engine). A game engine $\mathcal{E}$ is a *DECS-conformant engine* if and only if:

(1) It maintains a runtime state $\sigma = (E, M, Q, \Gamma, \varphi)$ as defined in Section 3;
(2) It executes the game loop δ according to the operational semantics of Section 3, preserving event FIFO order, rule list order, and entity iteration order;
(3) It implements the action semantics of Table 1 for all action types;
(4) It translates hardware input events through the abstract input set I into abstract events and enqueues them to Q.

Conformance requires the absolute preservation of the evaluation order, the atomicity of the transition δ, and the evaluation invariants defined in Section 3. A conformant engine is not bound to a specific serialization format; it is required only to map its chosen input schema (e.g., JSON, YAML, or multi-file structures) to the formal 5-tuple G such that the execution invariants are maintained. Because DECS descriptions are pure structured data, executability can be verified at load time by comparing the description's component references against the engine's declared capabilities. **The validity of a DECS game description does not depend on any specific conformant engine.**

DECS specifies a computational model, not a rigid file schema. A conformant engine may include *semantically inert metadata* in its serialization (such as editor-only labels or debug affordances) without breaking compliance, provided these fields do not alter σ or δ. Two DECS artifacts are *semantically equivalent* if they produce identical state sequences $\langle \sigma_0, \sigma_1, \ldots \rangle$ under the same input sequence, regardless of serialization format (JSON, YAML, or multi-file). The declarative-reactive structure constrains the space of valid representations sufficiently to enable systematic translation between conformant schemas, making execution strategies such as scene-based streaming or optimized loading possible without altering core semantics.

4.2. Reference Implementation

To validate executability, we implemented a minimal conformant engine in JavaScript running in any modern web browser. It is scoped to 2D games as a deliberate proof-of-concept boundary, and available as open-source software at `https://github.com/`

LUDES-PESC/DECS. It serves strictly as *proof of executability*, providing evidence that the declarative model is realizable and that the case-study descriptions execute correctly. It is not presented as a production-grade engine or canonical implementation.

4.3. Reference BNF

The following BNF defines the abstract syntactic structure of a DECS game description, serving as a guide for conformant engine implementations. The reference engine realizes this grammar as JSON; other conformant engines may use any serialization format that preserves the 5-tuple $G = (T, S, I, \Gamma, s_0)$ defined in Section 3.

```
<Game>        ::= { templates:<TemplateSet>, scenes:<SceneSet>,
                    start_scene:<Id>, input:<InputSet>,
                    [<ContainerKey>:<ContainerData>]* }
<TemplateSet>  ::= { <Id> : <Template> }*
<SceneSet>     ::= { <Id> : <Scene> }*
<Template>     ::= { components: <ComponentMap> }
<ComponentMap> ::= { <ComponentName>:<ComponentData> }*
<Scene>        ::= { width:N, height:N, camera:<Camera>,
                     entities:<EntityInstances>,
                     gravity?:Float, use_gravity?:Bool }
```

The Behavior component rule structure in JSON:

```
<Behaviors> ::= { <EventName> : [<Rule>*] }*
<Rule>      ::= { conditions?:[<Condition>*],
                  actions:[<Action>+],
                  probability?:Float in (0,1] }
<Condition> ::= { source:<Source>,
                  data:{ <Key>:<PredicateExpr> }*,
                  entity?:self|other }
<PredicateExpr> ::= <Value>
  | {">":<V>} | {"<":<V>} | {">=":<V>}
  | {"<=":<V>} | {"=":<V>} | {"includes":<V>}
<Action> ::=
  {type:set, target:<T>,component:C,key:K,value:V}
| {type:add, target:<T>,component:C,key:K,value:V}
| {type:multiply, target:<T>,component:C,key:K,value:V}
| {type:spawn, key:<TemplateId>,value:<SpawnParams>}
| {type:destroy, target:<T>}
| {type:emit, key:<EventName>,target:<T>,
              value?:<Payload>}
| {type:random, target:<T>,component:C,key:K,
                domain:<Domain>}
<Target> ::= self | other | broadcast
<Domain> ::= [<Value>*] | {min:<N>, max:<N>}
```

Note that the domain may be any finite set of values or a numeric interval, corresponding to the abstract D in Table 1.

The reference engine declares 21 systems as its capability set: Transform, Tag, Behavior, Timer, Attribute, Force, Movement, Jump, Inventory, Physics, Collision, Sensory, Hierarchy, Animation, Camera, Tilemap, Light, Dialogue, Text, Draggable, and Render. With this capability set the engine executed all case studies described in Section 5. Conformant engines targeting other domains would declare different capability sets.

5. Validation

We validate the DECS model through ten complete game descriptions in JSON, executed in the reference conformant engine. Table 2 lists all ten implementations. The breadth of

Table 2. All ten DECS-conformant game descriptions executed in the reference engine.

Genre / description	Key mechanics exercised
Physics puzzle	Rigid-body physics, drag-to-launch, HP-based destruction
Scrolling shooter	Multi-scene flow, entity hierarchies, procedural spawning
Brick-breaking game	Tilemap collision, per-tile state, multi-device input
Endless runner	Gravity, procedural gap generation, instant-death collision
Side-scrolling platformer	Jump mechanics, side-scroll camera, collision layers
Paddle game	Paddle AI, ball physics, score tracking
Grid-movement arcade	Discrete grid movement, growth mechanic, self-collision
Top-view RPG	Tile movement, inventory system, item crafting, dialogue
Music toy	Input-driven audio events, real-time UI interaction
Visual novel	Scene flow, branching dialogue, narrative state

covered genres includes non-traditional interactive software such as the piano and visual novel to support the claim that DECS is not inherently limited to any particular game genre or style.

Three of these games were selected for mechanical diversity and detailed with code excerpts below; Figure 2 shows screenshots of the three. The validity of each description is defined by the model, not by this particular engine.

Descriptions were authored constructively: new systems were added to the engine's capability set whenever a mechanic could not be expressed with existing ones, terminating when all ten were fully executable. This provides evidence that the declared 21-system repertoire is sufficient for the covered genres, following a strategy analogous to [Marín-Lora et al. 2020]. Consequently, the case studies validate the DECS *model* — that game logic expressible as declarative rules over a sufficient system repertoire requires no game-specific code — more directly than they validate any fixed capability set; a different conformant engine targeting the same genres could realize the ten descriptions with a different system repertoire, provided it satisfies the conformance criterion of Section 4.

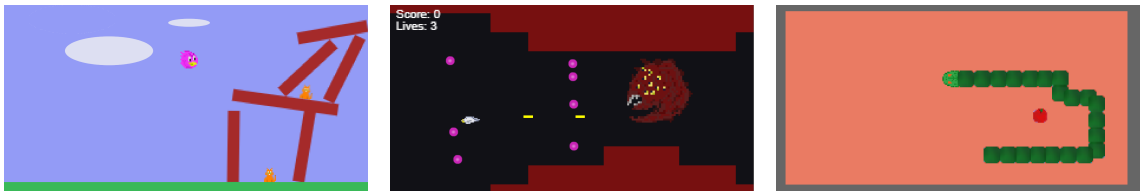

Figure 2. Screenshots of the three selected games.

5.1. Physics Puzzle

This case study describes a physics puzzle game (in the style of *Angry Birds*). This case study exercises rigid-body physics, tag-based entity discrimination, numeric threshold predicates, attribute arithmetic, and reactive entity destruction. One of its templates is the `blocker`, which implements hit-point-based destruction via four sequential `collision_enter` rules checking velocity thresholds, tag predicates, and HP state, demonstrating multi-factor reactive logic through data alone. It also uses a `bird` template that implements drag-to-launch via the `drag_end` event $ev \in Q$. Upon dequeueing in Step 3, the following rules are evaluated for $e \in \text{resolve\_target}(\texttt{self}, \ldots)$.

```
"drag_end": [{
  "actions": [
    { "type": "set", "component": "Physics", "key": "velocity_x", "value": "$vectorX"
        },
    { "type": "multiply", "component": "Physics", "key": "velocity_x", "value": 10 },
    { "type": "set",      "component": "Physics", "key": "use_gravity", "value": true
        },
    { "type": "set",      "component": "Draggable", "key": "enabled", "value": false
        }]}]
```

5.2. Scrolling Shooter

This case study describes a scrolling shooter (in the style of *Gradius*): six entity template categories, four scenes, global persistent state, timed procedural spawning, entity hierarchies, powerup pickups, and a boss encounter.

An invisible `enemy_wave` entity uses a repeating Timer to spawn enemies at randomized positions, while a child entity fires a deferred event after 30 s to trigger the boss encounter. Four scenes are connected via `load_scene` actions updating φ, with `lives` and `score` persisting in Γ across transitions. Weapon upgrades demonstrate state-dependent rule branching via $\Gamma[\texttt{powerup\_level}]$. This case study exercises multi-scene flow, global persistence, conditional branching, entity hierarchies, and deferred events.

```
"fire_start": [
  {"conditions": [{ "source": "Attributes",
      "data": { "weapon_mode": { "=": "normal" } } }],
    "actions": [{ "type": "spawn", "key": "bullet_h",
      "value": { "velocity_x": 300 } }]},
  {"conditions": [{ "source": "Attributes",
      "data": { "weapon_mode": { "=": "spread" } } }],
    "actions": [
      { "type":"spawn","key":"bullet_h",
        "value":{"velocity_x":300,"velocity_y":0}    },
      { "type":"spawn","key":"bullet_h",
        "value":{"velocity_x":300,"velocity_y":-100} },
      { "type":"spawn","key":"bullet_h",
        "value":{"velocity_x":300,"velocity_y":100}  }]}]
```

5.3. Grid-movement Arcade

This case study validates dynamic entity management and recursive state propagation in a *Snake*-style arcade game. The `grow` event demonstrates the modification of the entity set E via the `spawn` action $a \in \mathcal{A}$. A segment entity is instantiated with a `follow` behavior that uses resolve_src to track the position of the spawning entity in σ:

```
"grow": [{"conditions": [
    {"source":"Attributes", "data":{ "has_child":{ "=": false }}}],
  "actions":[
    {"type": "set", "target": "self", "component": "Attributes",
      "key": "has_child", "value": true },
    {"type": "spawn", "key": "snake_segment", "value": { "as_child": "segment" }}]}]
```

The `as_child` field is an engine-level parameter, not a component override in θ, illustrating how a conformant engine may extend abstract actions for its capability set without altering the model. State advances through discrete grid steps: each head movement emits a signal propagating through the tail as a recursive reactive chain. This case study also exercises dynamic E expansion and inter-entity attribute tracking.

6. Conclusion

We presented DECS (Declarative Entity-Component-System), a declarative model with explicitly defined operational semantics for describing complete, playable games as pure structured data, independently of any particular engine.

DECS' primary contribution is conceptual: the separation of game specification from game implementation is the same design move that gives virtual machine specifications, graphics APIs, and language standards their longevity. The specification becomes a stable contract around which multiple independent implementations can coexist, and a DECS game description is portable by construction across any conformant engine whose declared capability set covers its required mechanics. Any conformant engine can parse it and determine executability at load time, and one whose capability set covers the description's components can run it. Five concrete contributions follow from this foundation. First, a declarative model separates game *definition* from game *logic* through a Behavior component that relocates game-specific logic out of imperative code and into declarative reactive rules. Second, a complete semantic framework provides a runtime state model and small-step operational semantics for event-driven execution. Third, the Behavior component is formally characterised as a distributed production system with expressive power conjectured to be equivalent to register machines. Fourth, a conformance criterion decouples execution semantics from engine capability and enables load-time validation of executability. Fifth, empirical validation through ten complete game descriptions in JSON, executed in a reference conformant engine spanning diverse genres from physics puzzles to a visual novel, confirms the model is realizable without additional game-specific code beyond its declared capability set.

Code-free authoring holds relative to the declared system repertoire, and defining a minimal standard repertoire remains an open problem. The ten case studies offer initial evidence by spanning diverse 2D genres, though they are limited to offline, single-player settings. The DECS state model currently has no built-in notion of network synchronization or server-driven content updates, making multiplayer and live-service games a natural direction for future work alongside AI-driven, open-world, and 3D extensions. On the formal side, mechanized verification of the register machine reduction and controlled user studies comparing DECS authoring with scripting-based approaches remain to be conducted, leaving quantitative usability claims open [Voisard et al. 2025].

DECS pushes data-driven game development to a principled limit, treating any game as a portable, formally defined, engine-independent artifact.

References

Bayliss, J. D. (2022). Developing games with data-oriented design. In *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering (GAS '22)*, pages 30–36. ACM.

Brownston, L., Farrell, R., Kant, E., e Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley.

da Silva, P., Campos, R., e Rocha, C. (2021). OSS scripting system for game development in Rust. *IFIP Advances in Information and Communication Technology*, 624:51–58.

Ebner, M., Levine, J., Lucas, S. M., Schaul, T., Thompson, T., e Togelius, J. (2013). Towards a video game description language. In Lucas, S. M., Mateas, M., Preuss, M., Spronck, P., e Togelius, J., editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Fischbach, M., Wiebusch, D., e Latoschik, M. E. (2017). Semantic entity-component state management techniques to enhance software quality for multimodal VR-systems. *IEEE Transactions on Visualization and Computer Graphics*, 23(4):1342–1351.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37.

Johansen, M., Pichlmair, M., e Risi, S. (2019). Video game description language environment for unity machine learning agents. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

Julia, C. e van Rozen, R. (2023). Scriptbutler serves an empirical study of puzzlescript: Analyzing the expressive power of a game dsl through source code analysis. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*. ACM.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., e Genesereth, M. (2006). General game playing: Game Description Language specification. Technical Report LG-2006-01, Stanford Logic Group.

Marín-Lora, C., Chover, M., Sotoca, J. M., e García, L. A. (2020). A game engine to make games as multi-agent systems. *Advances in Engineering Software*, 140:102732.

Martens, C. (2015). Ceptre: A language for modeling generative interactive systems. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 11(1):51–57.

Minsky, M. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall.

Piette, E., Soemers, D. J. N. J., Stephenson, M., Sironi, C. F., Winands, M. H. M., e Browne, C. (2020). Ludii: The ludemic general game system. In De Giacomo, G., Catala, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., e Lang, J., editors, *ECAI 2020: 24th European Conference on Artificial Intelligence, Including 10th Conference on Prestigious Applications of Artificial Intelligence, PAIS 2020, Proceedings*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418. IOS Press.

Politowski, C., Petrillo, F., Montandon, J. E., Valente, M. T., e Guéhéneuc, Y.-G. (2021). Are game engines software frameworks? A three-perspective study. *Journal of Systems and Software*, 171:110846.

Pérez-Liébana, D., Liu, J., Khalifa, A., Gaina, R. D., Togelius, J., e Lucas, S. M. (2019). General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214.

Raffaillac, T. e Huot, S. (2019). Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proceedings of the ACM on Human-Computer Interaction*, 3(EICS):8:1–8:22.

Schaul, T. (2013). A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8. IEEE.

Thielscher, M. (2010). A general game description language for incomplete information games. In Fox, M. e Poole, D., editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 994–999, Atlanta, Georgia, USA. AAAI Press.

Thielscher, M. (2017). Gdl-iii: A description language for epistemic general game playing. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 1276–1282. International Joint Conferences on Artificial Intelligence Organization.

Voisard, L., De Freitas Serra, H., Politowski, C., Petrillo, F., e Guéhéneuc, Y.-G. (2025). A mapping study of the entity component system pattern. In *Proceedings of the 2025 IEEE/ACM International Workshop on Games and Software Engineering (GAS '25), co-located with ICSE 2025*.

Wiebusch, D. e Latoschik, M. E. (2015). Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 25–32.