Gaming Digital Twin Architectures for Reusability, Maintainability, and Evolvability

Kevin Bouwmeester[1], Michel R. V. Chaudron[1] and Loek Cleophas[1,2]

[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Stellenbosch University, Stellenbosch, Republic of South Africa

Abstract

This position paper proposes a software architecture for digital twins (DTs). We show how the Entity Component System (ECS) pattern used in this architecture enhances maintainability. To support this claim, we identify a variety of change-scenarios that apply to DTs and show how they are supported by our architecture. Through these scenarios we illustrate how the ECS pattern addresses software quality attributes relevant to evolution. Our architecture design is in essence generic, thus easing its (re)use across applications and domains.

Keywords

Digital Twins, Digital Twin Engineering, Digital Twin Architecture, Architecture patterns

1. Introduction

Digital Twins (DTs) are becoming increasingly popular in many domains, ranging from Cyber-Physical Systems (CPSs) and building management to the manufacturing and biomedical domains. Many recent publications have defined architectures or frameworks for DTs [1, 2]. However, these frameworks and architectures are often specific to their application domain, or do not explicitly discuss their re-usability beyond their specific domain. Consequently, (re-)using architectures from different domains is difficult, as the architectures often mix conceptual arguments with implementation details and are either created entirely in the technological space, do not support cross-domain integration, or both.

In the context of engineering of DTs, there is a need for reusable and easy to use architectural frameworks [3]. In addition, the evolution of DTs is challenging. Adding objects or services to an existing system requires appropriate changes [4]. Different frameworks or architectures might influence how this evolution takes place. However, only very few publications specifically discuss extensibility, re-configurability, re-usability, or scalability, when they present software architectures. When studies do discuss architecture, these software qualities are often a secondary concern [2].

In this paper, we demonstrate how the ECS pattern can be integrated in the architecture of DTs. The ECS pattern is a data-oriented software pattern originating from game design. The pattern is claimed to benefit extensibility, re-configurability, re-usability, and scalability [5, 6, 2], due to the decoupling of data from behavior. At the same time, the ECS pattern is known to help create highly performant Realtime Interactive Systems (RISs) [7], crucial for DTs as well. Although there are publications investigating the ECS pattern [5, 8, 9], these do not discuss how the software qualities are addressed by it. In this work, our aim is to explicitly discuss the software qualities related to evolution with respect to the ECS pattern. In addition, we will show how the ECS pattern could fit in the architecture of a DT system. Instead of defining behavior on an object level, the ECS pattern models behavior on a universal level, applying it to any objects that have the right properties. We claim that this approach of modeling behavior at universe level can improve the separation of concerns between domain modeling and technical implementation.

The remainder of this position paper is structured as follows. In Section 2 we provide a brief introduction on the ECS terminology and describe why the pattern is a good fit for DTs. In Section 3 we discuss existing work on ECS and DTs, while in Section 4 we present the system we developed

BENEVOL2026: Belgium-Netherlands Software Evolution Workshop 26–27 November, 2026, Madrid, Spain
k.bouwmeester@tue.nl (K. Bouwmeester); m.r.v.chaudron@tue.nl (M. R. V. Chaudron); l.g.w.a.cleophas@tue.nl (L. Cleophas)
0009-0002-5252-962X (K. Bouwmeester); 0000-0001-7517-6666 (M. R. V. Chaudron); 0000-0002-7221-3676 (L. Cleophas)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

following ECS. In Section 5 we discuss how the design presented in Section 4 relates to the software qualities relevant to evolution. Section 6 concludes the paper and discusses future work.

2. ECS pattern revisited

The ECS pattern is built around *Entity*, *Component*, and *System* concepts. Formalizations of the pattern exist, though these are very recent [10, 11]. An *Entity* relates to any labeled object that can exist in the system, and is commonly represented by an identifier only. Next, a *Component* refers purely to a data element that an Entity can have. Typically, multiple *Component* types exist in any ECS System. Component instances can be related as a bag, where the Entity represents the identifier of this bag.

In the ECS pattern, the parts called *System* define functionalities. The operational semantics of an ECS System is as follows: a *System* retrieves the collection of Component instances of which the types match its signature. Then, the *System* applies its functionality to all bags of Component instances in this collection - thereby updating the state of the Entity. This is much like the *map* construct in functional programming. Typically, the ECS continuously cycles through the above steps. As a result, if an *Entity* has particular *Component* types, it will exhibit the behavior that results from the repeated application of the *System*-functionality that applies to these Component types.

The concepts of the ECS pattern map very well to the key concepts of DTs. Entities correspond to the objects, Components to the properties, and Systems to the analysis or behavior of interest. Moreover, cyclical execution of services aligns with iterative incremental steps in DT simulations.

3. Related Work

Hatledal et al. [8] present Vico, a high-level co-simulation framework that aims to support the Functional Mock-up Interface (FMI) standard [12]. Its use of ECS sets it apart from other co-simulation frameworks. It has been designed for flexibility, which does come with downsides, as there are no pre-defined rules defining how simulations should be designed. However, the work by Hatledal et al. does not explicitly discuss how the ECS pattern addresses the evolvability-qualities. As such, we believe our work complements it, as in our work we intend to discuss the ECS pattern in more detail. As a result, our work could provide insight on why Vico could be beneficial over other co-simulation frameworks.

Cheng et al. [5] created ECS-Grid, a real-time simulation platform for Cyber-Physical Power Systems (CPPSs). They later used this for a DT for low earth orbit satellite constellations [9]. The authors describe ECS concepts at high level, but then quickly focus on implementation details. As a result, it is unclear whether the advantages they describe are a result of their engineered simulation platform, or of their use of ECS. In contrast, we provide a more conceptual analysis of the pattern, explaining its principles with respect to use in DTs. This would illustrate how and why ECS-Grid has the advantages indicated by Cheng et al. Furthermore, their work leaves applicability beyond CPPSs unclear.

4. A System based on ECS

We discuss the high-level architecture of the system developed following the ECS pattern. We exemplify it through a case-study, revolving around the simulation component of a DT for Earth's satellite network. More detailed information on the design, as well as the source code, can be found on GitHub[1].

Our system is built on a generic 4-layer reference architecture to which we map the components introduced by ECS. The bottom layer is the *Actual System Layer*, which holds the software components that allow the system to interface with sensors and actuators. On top of it lies the *Common Data Layer* with components related to the storing of data. Next we define the *Application Services Layer* with components related to reasoning, simulation, and other services that act upon the common data. Finally we define a *Visualization & Interaction Layer* with components aimed at human-computer interaction.

[1] https://github.com/xhilusius/DT-ECS

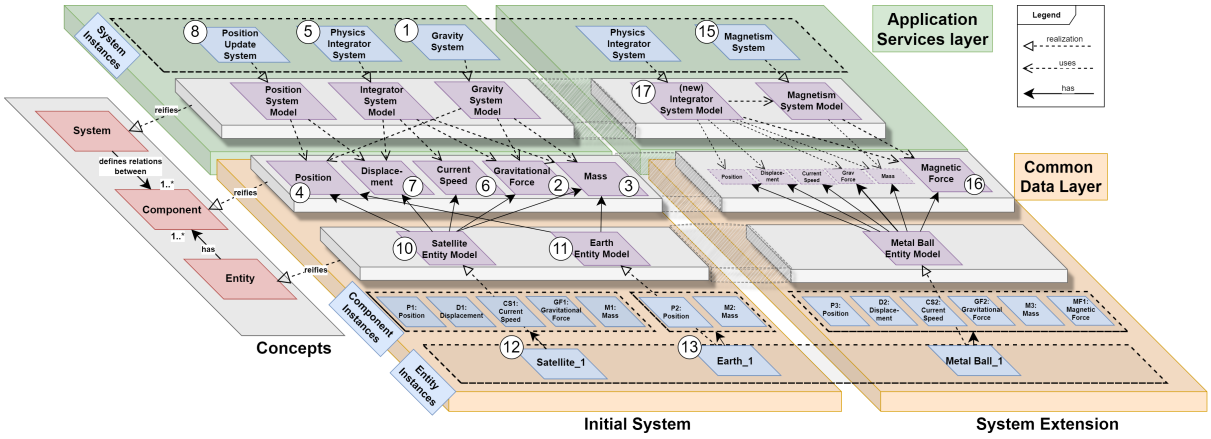


Figure 1: A diagram showing the mapping between the primary concepts (red), the domain models (purple), and an example instance (blue), on 2 layers of a reference architecture.

Given the scope of the ECS pattern, we focus on the *Common Data Layer* and *Application Services Layer*. In Figure 1 we show a visual overview of these layers, as well as the mapping of the different ECS components to each layer. The different colors in the figure refer to different levels of abstraction of the elements, where purple represents high level models, and blue their instantiations. In the figure, we illustrate the terms (shown in red), each related to the concepts of ECS. The first step involves reifying these concepts, defining what *Entities*, *Components*, and *Systems* we want to model.

In our example, we aimed to create a DT for the satellite network in earth orbit. The simulation is used to execute *what-if* scenarios to see whether a collision would occur within a certain time when introducing a new satellite to earth's orbit. To simplify the simulation, we focus on modeling the gravitational pull of the Earth on its satellites.

Each System model is a reification of some domain model, as it defines how specific domain properties are related to other properties. For example, consider Newton's law of universal gravity, $F = G * \frac{m_1 * m_2}{d^2}$. This introduces the need for Components that represent the variables in the equation. We can define a conceptual execution flow as follows. First, some *Gravity System* (1) would calculate the *Gravitational Force* (2) of entities, using their Mass (3) and Position (4). The System will use all entities that have a mass and position, to calculate the gravitational force for entities that have all 3 Components. Next, some *Physics Integrator System* (5) will take the *Gravitational Force*, together with the *Mass* and *Current speed* (6), to calculate a *Displacement* (7). Finally, a *Position Update System* (8) calculates the new *Position* of each entity using their current *Position* and the calculated *Displacement*.

Given the set of Systems and Components, we can now model the Entities. For an Entity to exhibit gravitational behavior, it requires the right Components. In case we want to model *Satellites* (10) orbiting *Earth* (11), each satellite instance would need to have a position, displacement, current speed, gravitational force, and mass Component instance. Instantiating the Earth Entity with the same Components results in it exhibiting the same behavior. In Figure 1 we illustrate a case where the Earth only has a position and mass. As a result, it is included in the gravitational force calculations of each satellite, but does not move itself, since it lacks other Components.

5. Discussion

Now that we have discussed and exemplified the use of the ECS pattern in the architecture and design of DTs, we consider how it deals with the software qualities related to evolution mentioned before.

5.1. Re-configurability

For re-configurability, we refer to the changing of existing Entities, Components, or Systems.

Reconfiguration of Entities: Due to the focus on composition over inheritance, Entities do not have any inherited definition. What Entities represent and how they behave depend entirely on the Components with which they are composed. As such, it is possible to freely reconfigure an Entity at runtime.

Any Entity can have any Component, even if this results in uncommon, or incorrect, behavior. This can be observed in the instantiation of the *Satellite_1* (12) and *Earth_1* (13) Entities shown in Figure 1. In the example, *Earth_1* would not be subject to movement, although the System does include it in calculations on the gravitational pull it applies to other Entities. If we would add an Entity to represent the sun similarly to *Earth_1*, neither would move, creating an unrealistic simulation. As such, it is important to apply composition constraints if a user is unaware of the validity of Entity compositions.

Reconfiguration of Components: Components act as an interface between Entities and Systems, so reconfiguring them might affect any System using them. The value of a Component instance can be changed at runtime. However, changing the data type used for the Component requires updating every System that uses it. Changing the underlying model of the Component, i.e., changing the definition of a Component, requires completely updating any System that uses it. If for example the *Mass* Component shown in Figure 1 (3) changes from grams to kilograms, the *Physics Integrator System* (5) would need to be updated. Removing a Component type would require changes in the core of the System models themselves. If the *Displacement* (7) Component were removed, the *Position System Model* would need to be adjusted, as it would need to use other Components to calculate *Position* values.

Reconfiguration of Systems A System can be reconfigured freely as long as this reconfiguration remains consistent with the interface of the previous version. In other words, we can change how a System model is implemented as long as it still processes the same Components. However, this depends on whether other Systems depend on these Components. In our work, we experimented with this, by changing the implementation of the *Gravity System* (1). For example, we can simplify the gravity equation by combining Newton's law of universal gravitation with Newton's second law of motion. This results in the equation $F = g * m$, which does not require the distance between Entities to be used, as it only calculates the pull Earth has on Entities on its surface. However, since it still updates the same *Gravitational Force* Component, no dependency issues were observed. As System execution is expressed as a workflow, where one System executes before another, it is crucial that the Components required by later Systems do not change. This shows that System implementation can be reconfigured as long as Component dependencies are taken into account; Systems do not depend on the implementation of other Systems but on the Components that these Systems use.

5.2. Extensibility

Extensibility refers to adding new *Entities*, *Components*, or *Systems*.

Extending Entities: The possibility of adding new Entities is a key quality of the ECS pattern. Entirely new Entity compositions can be made, leading to new types of Entities in a system. Adding a new Entity, and composing it with some Components, will result in it having its Component instances processed in the next System execution loop iteration, assuming its Components match a Systems contract.

Instances of new Components can be added to the composition of Entities, regardless of whether a System processes them, as this is the same as reconfiguring the Entity. If a new Component is added to an existing System, it implies the model of this System has been changed. If this addition does not affect the meaning of the other Components in the System, e.g. as with the change to the *Gravity System* no longer using the *Position* Component, there are no dependency issues.

Extending Systems: As Systems might have indirect dependencies to other Systems, it is important to closely document these indirect dependencies. In Figure 1, we illustrate a system extension that also introduces an additional Component. After having the gravity System implemented, we wanted to add a simplistic System that would mimic a magnetic force originating from a fixed location. The *Magnetism System* (15) was defined for this purpose. The System calculates a *Magnetic Force* (16) Component value based on the Entity's *Mass* and a fixed magnetic force constant. The System is independent from the *Gravity System*, and as such can be executed in parallel. However, since the *Displacement* is calculated based on the force vector of the Entity, it required an update to the *Integrator System Model* (17). If

only one force Component was part of some Entity, it would use only that to calculate the resulting *Displacement*. If an Entity has both a *Magnetic Force* and a *Gravitational Force*, the System uses the sum of these instead to calculate the *Displacement*. Consequently, since the unchanged *Displacement* Component couples the *Physics Integrator System* and the *Position Update System*, no changes were required to the latter.

In short, adding new Systems requires the workflow defining the execution order to be revisited. It is important to observe whether the System is unrelated to other Systems, as this could indicate opportunities for parallelization. If a new System uses existing Components, this provides an indication of where the new System should be placed in the execution order. If the new System modifies instances of an existing Component, this might result in dependency issues that need to be resolved. Worst case, this might require any Systems that depend on the same Component to be updated.

5.3. Re-usability

Re-usability primarily refers to the reuse of elements for different DT projects or systems, either in similar or different domains. As Entities have no inherent definition, their reuse is meaningless without reusing Components. Components can be freely reused, similar to reusing a data schema.

Reusing Systems is only possible together with the Components on which they depend. Since there is no direct dependency between the definitions of Systems and Entities, System models and their implementations can easily be reused to fit different systems or domains. In our case, our initial system was aimed at simulating falling balls. The *Gravity System* that was developed for this, which uses the equation of Newton's law of universal gravitation, was reusable for our satellite simulation.

In a practical setting, this could improve the reuse of such Systems between domains that share similar phenomena, or desired Systems. A System that simulates fluid dynamics can be used in both the Biomedical (blood flow) and civil engineering domains (water transport).

5.4. Scalability

Scalability refers to the possibility of setting up a system that deals with many Entities, Components, and Systems. The performance of a single System is limited by its number of Component instances. The ECS pattern promotes storing all instances of a single Component in a sequential array. The fetching of data arrays for Systems is therefore optimized; however, ensuring all such Component instances can be stored in memory or cache, would improve performance, as it would reduce the need to fetch data. Optimizing code by using parallelization and specific hardware can also improve vertical scalability.

Systems are decoupled, and a dependency is the result of Systems using the same Component. It is possible to execute multiple Systems in parallel, given they are independent. Although execution is orchestrated following some execution flow, each System could be executed on its own computational node. As a result, horizontal scaling can be achieved by distributing computation, with the execution of a System done by specialized nodes. In general, systems using the ECS pattern display similarly high scalability as Service Oriented or Microservice architectures. In addition to loose coupling between Systems, it also promotes the storage of state data in contiguous blocks, which optimizes cache efficiency.

6. Conclusion & Future Works

In this work, we presented a software architecture using the ECS architectural pattern. We exemplified this architecture through a case study, where we analyze various scenarios to show how the ECS pattern addresses quality attributes of the software relevant to evolution.

We intend to investigate reuse of components from existing domain ontologies or knowledge graphs, providing traceability with known domain concepts.

We also aim to allow domain experts to model their Systems, and their execution flow, using higher levels of abstraction, where implementation details are hidden. We believe that there are possibilities to support this using low-code tooling, though this requires further investigation.

References

[1] R. Eramo, F. Bordeleau, B. Combemale, M. van den Brand, M. Wimmer, A. Wortmann, Conceptualizing Digital Twins, IEEE Software 39 (2022) 39–46. doi:10.1109/MS.2021.3130755.

[2] E. Ferko, A. Bucaioni, M. Behnam, Architecting Digital Twins, IEEE Access 10 (2022) 50335–50350. doi:10.1109/ACCESS.2022.3172964.

[3] F. Bordeleau, B. Combemale, R. Eramo, M. van den Brand, M. Wimmer, Towards Model-Driven Digital Twin Engineering: Current Opportunities and Future Challenges, in: Ö. Babur, J. Denil, B. Vogel-Heuser (Eds.), Systems Modelling and Management, Springer International Publishing, Cham, 2020, pp. 43–54. doi:10.1007/978-3-030-58167-1_4.

[4] J. Michael, L. Cleophas, S. Zschaler, T. Clark, B. Combemale, T. Godfrey, D. E. Khelladi, V. Kulkarni, D. Lehner, B. Rumpe, M. Wimmer, A. Wortmann, S. Ali, B. Barn, I. Barosan, N. Bencomo, F. Bordeleau, G. Grossmann, G. Karsai, O. Kopp, B. Mitschang, P. Muñoz Ariza, A. Pierantonio, F. A. C. Polack, M. Riebisch, H. Schlingloff, M. Stumptner, A. Vallecillo, M. van den Brand, H. Vangheluwe, Model-Driven Engineering for Digital Twins: Opportunities and Challenges, Systems Engineering 28 (2025) 659–670. doi:10.1002/sys.21815.

[5] T. Cheng, T. Duan, V. Dinavahi, ECS-Grid: Data-Oriented Real-Time Simulation Platform for Cyber-Physical Power Systems, IEEE Transactions on Industrial Informatics 19 (2023) 11128–11138. doi:10.1109/TII.2023.3244329.

[6] L. Voisard, H. De Freitas Serra, C. Politowski, F. Petrillo, Y.-G. Géhéneuc, A Mapping Study of the Entity Component System Pattern, in: 2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS), IEEE, Ottawa, ON, Canada, 2025, pp. 33–40. doi:10.1109/GAS66647.2025.00010.

[7] D. Wiebusch, M. E. Latoschik, 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), IEEE, Arles, France, 2015, pp. 25–32. doi:10.1109/SEARIS.2015.7854098.

[8] L. I. Hatledal, Y. Chu, A. Styve, H. Zhang, Vico: An entity-component-system based co-simulation framework, Simulation Modelling Practice and Theory 108 (2021) 102243. doi:10.1016/j.simpat.2020.102243.

[9] T. Cheng, T. Duan, V. Dinavahi, Real-Time Cyber-Physical Digital Twin for Low Earth Orbit Satellite Constellation Network Enhanced Wide-Area Power Grid, IEEE Open Journal of the Industrial Electronics Society 5 (2024) 1029–1041. doi:10.1109/OJIES.2024.3454010.

[10] P. Redmond, J. Castello, J. M. Calderón Trilla, L. Kuper, Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern, Proceedings of the ACM on Programming Languages 9 (2025) 1–28. doi:10.1145/3763050.

[11] A. Tasnim, T. Zhao, The Essence of Entity Component System, in: Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing, SAC '26, Association for Computing Machinery, New York, NY, USA, 2026, pp. 1320–1329. doi:10.1145/3748522.3779910.

[12] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, A. Viel, Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models: 9th International Modelica Conference, in: Proceedings of the 9th International Modelica Conference, The Modelica Association, Munich, Germany, 2012, pp. 173–184. doi:10.3384/ecp12076173.