Unified Patterns for Realtime Interactive Simulation in Games and Digital Storytelling

Dieter Schmalstieg

Graz University of Technology

***Abstract*—This paper discusses the state of the art in realtime interactive simulation patterns, which are essential in the development of game engines, digital storytelling, and many other graphical applications. We show how currently popular patterns have evolved from equivalent, or, at least, very similar concepts, and are deeply related. We show that game engines do not need to choose a particular pattern, but can easily combine all desired properties with little or no overhead.**

Realtime interactive simulations, such as computer games or cosimulations,[1] typically use *entities* and *tasks* as their building blocks. A game world or other graphical environment is populated by entities such as creatures or treasures. Many entities coexist, over longer periods of time, and the digital narrative emerges as a consequence of their simulated interactions. The challenge from a programming point of view is scalability: We want to make it convenient for the programmer to express the desired simulation in terms of entities and tasks, and the simulation should be flexible in accommodating new entities and new tasks.

Contemporary game engines, such as Unity3D, favor the *entity/component/system* (ECS) pattern,[2] which makes it easy to extend entities and tasks independently. ECS decouples entities and tasks, which is often seen as a major advantage over traditional object-oriented programming techniques. However, it does not support communication between entity *groups*. Mercury[3] has recently shown how Unity3D can be extended with group communication based on dataflow, but Mercury still requires to know communication patterns in advance.

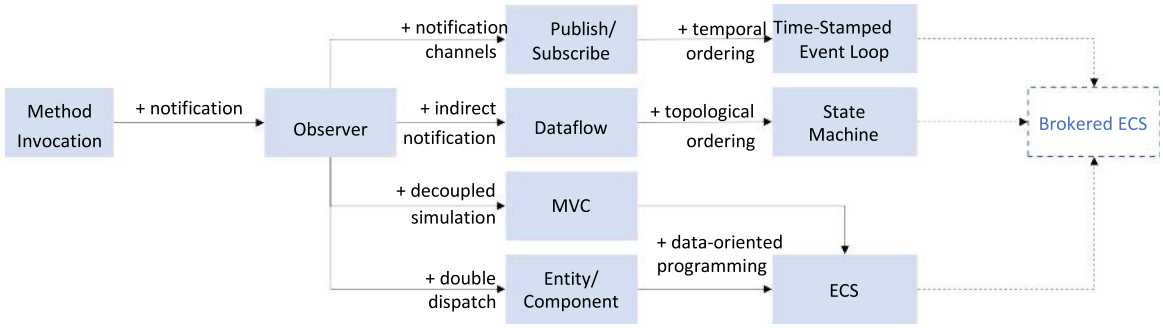

Figure 1. Relationship of the architectures (blue boxes) discussed in this article. Arrows denote the addition of a particular concept.

In this article, we compare several popular communication patterns and show how the combination of ECS with dynamic group communication unifies all previously proposed patterns into a coherent whole (Figure 1).

NOTIFICATION

In object-oriented languages, entity types are typically derived from an abstract base class, while tasks are implemented as entity methods. Direct method invocations implies that the invoking entity must frequently poll the invoked entity to learn about any new information. Polling is wasteful, if there are many entities or infrequent changes. Moreover, the *schedule* in which tasks are carried out is bound to the invocation sequence. We could inadvertently trigger a deep sequence of method invocations, starving more urgent tasks.

Consequently, we replace polling with *notification*: Entities communicate by passing messages about *events* that happened in the simulated world. In order to receive events, an *observer* entity registers itself with a *subject* entity to receive notifications.[4] Notification separates event creation and consumption. Notifications can be delivered in a different order than the one in which the corresponding events were created. This freedom of reordering notifications enables the development of a large variety of scheduling strategies, suiting the needs of diverse flavors of simulation.

DATAFLOW

The observer pattern makes it easy to control notification propagation. A single observer can register its interest with many subjects, and a subject can receive interest from many observers. Moreover, multiple observers can be arranged in sequence, allowing indirect notification, provided that events are automatically forwarded to the next observer in the chain. The resulting structure is a *dataflow* graph (or pipes-and-filters pattern),[5] a directed graph with entities as nodes and subject-observer relationships as edges.

A naive dataflow schedules nodes in depth-first order, as with direct method invocations. Alternatively, a more sophisticated scheduling can interpret the nodes as states in a *state machine*. Notification starts at a source node and is propagated based on state machine rules (see Figure 2). For example, breadth-first scheduling would require that all nodes with a shorter distance to the source must be processed first, before a given node may be processed. More complex rules can impose arbitrary *topological ordering* on the notifications. For example, one may require that a node has events pending on some or all of its incoming connections. Such a flexible scheduling can be implemented using a *visitor* pattern,[4] an iterator which traverses a graph according to a rule set and delivers notifications by invoking a virtual function on each visited node.

Dataflow has been used in many interactive applications. Visualization or image processing pipelines can be set up as dataflow,[6] representing filters, transformations, and encodings as nodes. Input processing in virtual and augmented reality is often specified as dataflow,[7] to stream input from multiple sensors through filters, fuse operations, and apply geometric transformations.

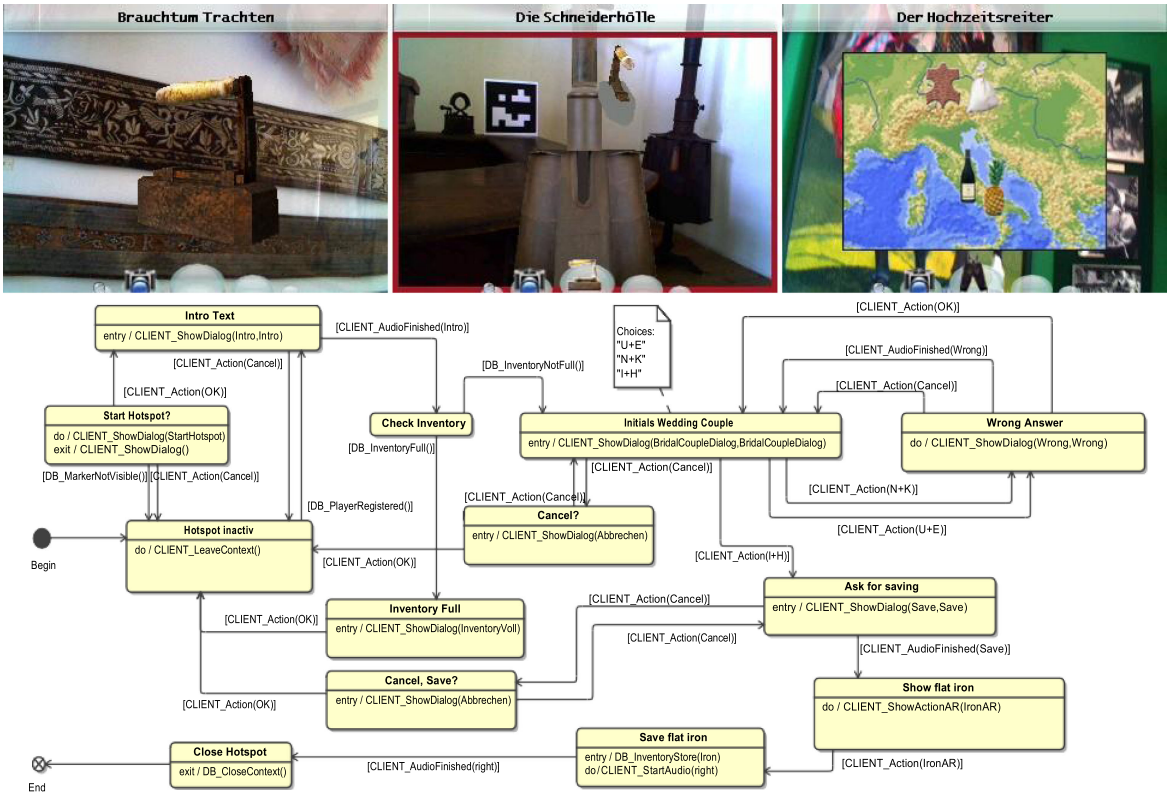

Figure 2. (Top) Augmented reality in a folklore museum. Left: The visitor retrieves the flat iron for finding out the initials of the bridal pair; Middle: The visitor heats the iron in the tailor's oven. Right: The visitor learns about the "wedding rider" custom. (Bottom) Finite state machine expressing the application logic of the folklore museum experience.

A *scene graph,*[8] a common data structure for visualization and games, can be interpreted as a special case of dataflow (see Figure 3). The visual representation of a scene naturally lends itself to a hierarchical representation, with nodes representing scene objects (triangles, textures, shaders, etc.) or geometric transformations. A visitor traverses the nodes of the scene graph, accumulates a transformaton matrix and calls a rendering method on each node.

PUBLISH/SUBSCRIBE

Dataflow supports indirect notification, but still requires explicitly setting up the links between entities to send notifications between entities. Setting up links for static relationships is easy, but, if entities act autonomously, associations change in dynamic, unpredictable patterns.

In such cases, we would like to specify entities to be notified by filtering attributes or specifying interests, rather than by enumerating entities directly. Instead of a visitor, we need a *publish/subscribe* architecture:[5] Entities publish to a notification *channel*, while other entities subscribe to this channel. The subscription can be determined by explicitly associating entities with channels, but a more powerful scheme determines subscription based on dynamic filtering of attributes, such as proximity. An intermediary object, the *broker*, collects events and delivers notifications. The broker can implement arbitrary scheduling strategies and even provide persistent storage.

When the broker collects and buffers many events from many entities, *temporal ordering* of events becomes essential. Proper temporal ordering requires that notifications for events are delivered in the half-order induced on the events' time-stamps.

MODEL/VIEW/CONTROLLER

If event processing and rendering are coupled together in the same loop, either one of the two tasks can become a bottleneck.

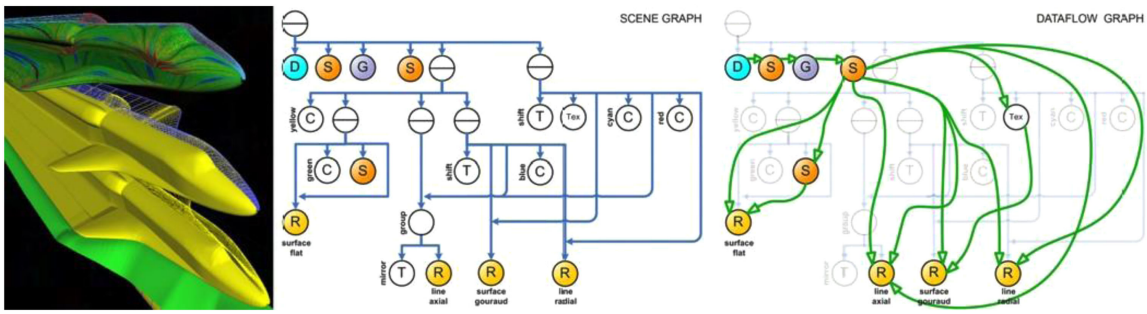

Figure 3. Flow visualization around a space shuttle (left) is modeled as a hybrid of scene graph (middle) and a dataflow graph (right).

Simulations should, therefore, incorporate a *decoupled simulation* pattern[9] that assigns separate threads of control to concurrent tasks. In a decoupled simulation, a broker may place its event queues in shared memory, accepting events from entities belonging to one thread and notifying entities belonging to another thread. Provided we use a strictly asynchronous scheduling, where no return notifications are required, each task may iterate over the received notifications at its own pace, without blocking other tasks.

In a simple architecture, a main thread may be responsible for brokering events, while other, secondary threads are spawned on demand for compute-intensive activities and retire after they have fulfilled their goal. In a more complex architecture, multiple threads may permanently tend to activities such as rendering, AI, animation, physics, and so on.

The *model/view/controller* (MVC) pattern[5] describes a common form of decoupled simulation. The *model* is a common store for the entities, while *controller* and *view* are tasks observing of the model. The view is responsible for rendering, while the controller is responsible for processing notifications. If needed, MVC can also use multiple views and multiple controllers per model.

ENTITY/COMPONENT/SYSTEM

While MVC can easily provide decoupled simulation of multiple tasks, a conventional visitor is designed for one specific task. The visitor needs to know which virtual method to call on the visited entities. If the set of tasks is not known in advance, or the task changes from node to node during visitation, a conventional visitor cannot be used. Such a case requires *double dispatch,*[10] the ability to vary both entity and task dependent on the use case. There are several ways how double dispatch can be implemented:

- *Multiple inheritance* can support double dispatch to some degree, but can lead to competition among inherited implementations ("deadly diamond of death").
- *Reflection* can extract a method signature from events and invoke a corresponding method. Reflection can either be natively supported by the language (e.g., C#) or emulated by parsing a textual event representation. Using reflection is generally considered too costly for processing large amounts of events.
- A *signal/slot* pattern can be used, which allows registering methods of the notified object to a particular signal (event). If the programming language does not support a signal/slot construct, it can be implemented by using templates or building a table of function pointers.
- The VIGO (views/instruments/governors/objects) pattern[11] extends MVC with a form of double dispatch where behaviors are split across two independently varied types of controllers, called "governors" and "instruments." Instruments represent tasks, while governors are mediators[4] facilitating appropriate reactive behaviors of entities.
- Inheritance can be replaced by *aggregation*.

The *entity/component* pattern[12] takes the latter approach, by aggregating components inside an entity. Each component is responsible for a

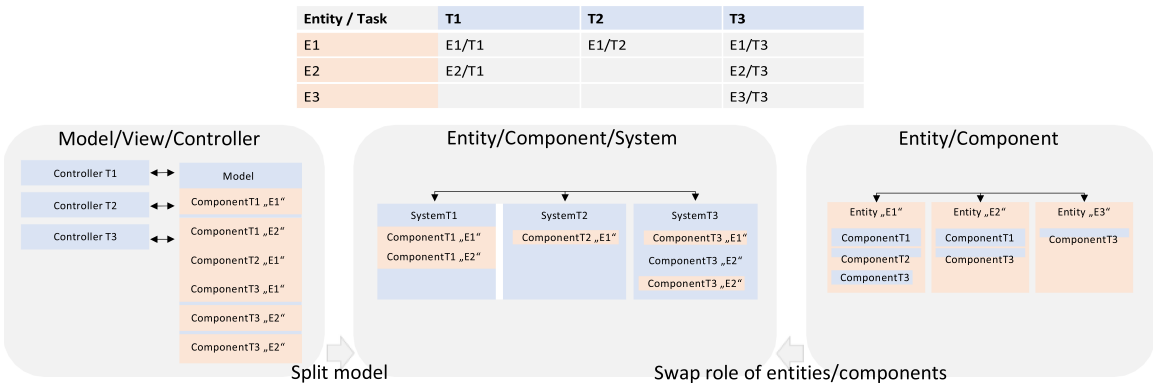

Figure 4. ECS pattern (bottom) can be explained as a combination of concepts from the model/view/controller (left) and the entity/component (right) pattern. In ECS, a given set of entities and tasks (top) is represented inside per-task systems.

specific task. Entities and components are derived from an entity base class and a component base class, respectively. Hence, implementations of entity and component can be varied independently to achieve double dispatch. When an entity receives an event, the entity searches its component directory for a suitable component and passes the event.

The entity/component pattern provides an elegant solution for double dispatch, but it is not ideal for data locality, since it organizes the data by an entity and not by task. Encapsulation of components inside entities implies that potentially very different data elements are aggregated inside one entity. A better data organization would result from following *data-oriented programming* principles and organizing data by task, not by an entity. This principle is known from column-oriented stores or the structure-of-arrays pattern.

We can combine the entity/component pattern with aspects of the MVC pattern to obtain the ECS pattern.[2] Entities become nothing more than a unique entity id. Tasks are split into a model part (the component) and a controller part (the *system*). A rendering system corresponds to the view of MVC.

Entities communicate by double-dispatch, as in entity/component, but with inverted order: first, on the system (task) level and, second, on the component (entity) level. Like in MVC, every system can run in its own independent thread, facilitating decoupled simulation. Inside a system, only components belonging to the same task are represented, leading to enhanced data locality if a task requires many entities to collaborate.

An alternative implementation of ECS can let systems handle events directly, rather than delegating the event to a component.[13] In this variant of ECS, the system more closely resembles the controller of MVC, by using components as mere data containers. The system cannot benefit from double dispatch afforded by a class hierarchy of components. However, using containers as pure data stores has distinct advantages. First, it becomes easier to use relational databases for persistent storage. Second, a task that needs to run intensive processing on a large number of identically structured components (e.g., finite element simulation) is more efficiently carried out on the system level without invoking virtual methods. Third, a system that just wraps around an existing library is essentially implementing a *mediator* pattern[4] and has no use for components.

BROKERED ECS

From the architectures discussed so far (see Figure 4), we arrive at three distinct patterns: (1) ECS is the preferred implementation pattern for modern game engines. (2) Publish/subscribe is a pattern often employed in networked systems, such as the internet of things or multiuser virtual reality. (3) Dataflow is commonly used in visualization and interaction. Mercury[3] combines ECS with dataflow,

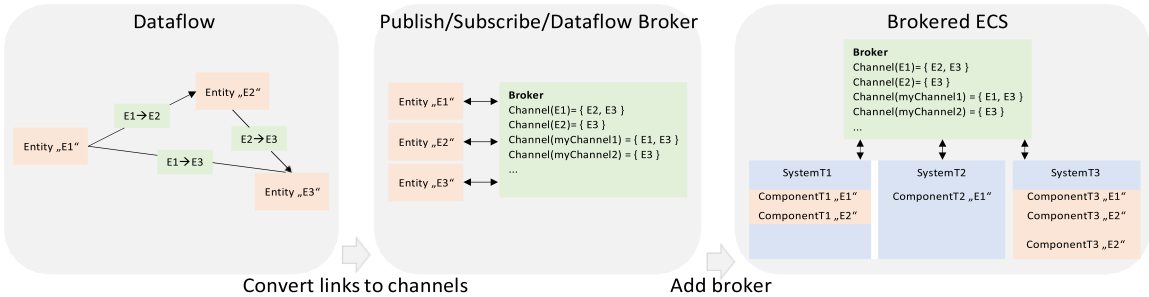

Figure 5. Dataflow architecture can be integrated into a publish/subscribe architecture by extending the broker. In the Brokered ECS variant, direct notification is replaced with the extended broker.

but lacks a more powerful publish/subscribe mechanism. This observation prompts the questions how difficult it would be to combine all three patterns.

Let us consider a combination of dataflow and publish/subscribe (see Figure 5, left). Obviously, notification can be sent to direct and indirect neighbors of an entity through a channel. The channel access may require an additional lookup compared to a callback mechanism, but if we assume at least one such lookup is required for using a virtual method, the overhead of going through a channel can be hidden with proper programming techniques. Moreover, an extended broker can enforce topological ordering by checking the constraints imposed on a node before an event is delivered. Hence, we can easily integrate dataflow, state machines and publish/subscribe.

Inserting such an extended broker into an ECS completes our system integration (see Figure 5, right). A *brokered ECS* routes notifications between components inside separate systems through a channel. Membership in channels facilitates arbitrary group notification. All forms of notification benefit from the double dispatch capability inherited from Entity/Component.

CONCLUSION

This paper establishes a unified view on several popular architectural patterns for game engines and other graphical simulations. We show that these patterns are not competing, but deeply related. Consequently, we arrive at an integrated approach, brokered ECS, which combines the desired properties of all discussed approaches: It supports all forms of notification, topological and temporal ordering, decoupled simulation, double dispatch, and data-oriented programming. Since a brokered ECS is only marginally more complex to build than the previous approaches, we conclude that making a hard choice between old and new programming style in realtime interactive simulation is not necessary.

REFERENCES

1. C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: State of the art,” CoRR, abs/1702.00686, 2017.
2. S. Bilas, “A data-driven game object system,” Blog Post, 2002. Available at: http://gamedevs.org/uploads/data-driven-gameobject- system.pdf
3. C. Elvezio, M. Sukan, and S. Feiner, “Mercury: A messaging framework for modular ui components,” in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2018, pp. 588:1–588:12.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.
5. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Hoboken, NJ, USA: Wiley, 1996.
6. W. Schroeder, K. M. Martin, and W. E. Lorensen. The Visualization Toolkit (4th Ed.): An Object-oriented Approach to 3D Graphics, 2006.
7. G. Reitmayr and D. Schmalstieg, “Opentracker - A flexible software design for three-dimensional interaction,” *Virtual Reality*, vol. 9, no. 1, pp. 79–92, Dec. 2005.

8. P. S. Strauss and R. Carey, “An object-oriented 3d graphics toolkit,” *SIGGRAPH Comput. Graph.*, vol. 26, no. 2, pp. 341–349, Jul. 1992.
9. C. Shaw, M. Green, J. Liang, and Y. Sun, “Decoupled simulation in virtual reality with the mr toolkit,” *ACM Trans. Inf. Syst.*, vol. 11, no. 3, pp. 287–317, Jul. 1993.
10. D. H. H. Ingalls, “A simple technique for handling multiple polymorphism,” in *Proc. Conf. Proc. Object-Oriented Program. Syst., Languages Appl.*, 1986, pp. 347–349.
11. Michel Beaudouin-Lafon Clemens Klokmose. “Vigo: Instrumental interaction in multi-surface environments,” in *Proc. ACM CHI*, 2009, pp. 869–878.
12. C. Stoy, “Game object component system,” in *Game Programming Gems 6*, 2006.
13. A. Martin, “Entity systems are the future of mmog development,” Blog Post, 2007. [Online]. Available: at: http://tmachine.org/index.php/2007/09/03/entity-systemsare- the-future-of-mmog- development-part-1/