Combining Discrete Event Simulations and Event Sourcing

Benjamin Erb
University of Ulm
Institute of Distributed Systems
benjamin.erb@uni-ulm.de

Frank Kargl
University of Ulm
Institute of Distributed Systems
frank.kargl@uni-ulm.de

ABSTRACT

Discrete event simulations (DES) represent the status quo for many different types of simulations. There are still open challenges, such as designing distributed simulation architectures, providing development and debugging support, or analyzing and evaluating simulation runs. In the area of scalable, distributed application architectures exists an architectural style called event sourcing, which shares the same inherent idea as DES. We believe that both approaches can benefit from each other and provide a comparison of both approaches. Next, we point out how event sourcing concepts can address DES issues. Finally, we suggest a hybrid architecture that allows to mutually execute simulations and real applications, enabling seamless transitions between both.

Categories and Subject Descriptors

I.6.0 [**Computing Methodologies**]: Simulation and Modeling—*General*; D.2.11 [**Software**]: Software Engineering—*Software Architectures*

General Terms

Design, Performance

Keywords

Discrete event simulation, event-driven architecture, event sourcing

1. INTRODUCTION

For both simulation systems and enterprise applications, the current values of its entities represent essential properties of the system over time. In discrete event simulations (DES), state is maintained for all simulated entities. Executing new simulation events incrementally modifies the simulation state and eventually generates a final outcome. Given a correct simulation model and simulation program, we can deduce results for the simulated domain and explore potential consequences.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Simutools 2014, March 17-19, Lisbon, Portugal
Copyright © 2014 ICST 978-1-63190-007-5
DOI 10.4108/icst.simutools.2014.254624

In enterprise applications, application state contains the current values of all domain objects and is usually modified by various triggers such as user interactions or internal occurrences. Enterprise systems read and modify state by accessing and altering values, in most cases backed by a database system. Within the last years, an alternative style for application architectures appeared and has been increasingly gaining attention, especially within distributed and cloud environments [1]. This style, called event sourcing, does not directly modify state by altering values. Instead, it stores the sequence of events that trigger state transitions. Event sourcing is actually sharing the same inherent idea as DES: an execution model represented by a sequence of events.

In this position paper, we show that it is promising for the simulation community to have a closer look at event sourcing and related concepts. We believe that both approaches can benefit from each other: Simulation engines can apply concepts related to event sourcing that might enhance the development process of simulations, facilitate debugging, and improve simulation analysis. Distributed simulation engines can adopt architectural concepts from event sourcing. On the other hand, enterprise architectures embracing event sourcing can take advantage from the decades of discrete event simulation research, such as parallelization efforts.

The remainder of this paper is structured as follows. First, we introduce event sourcing and related concepts for application architectures. Then we compare event sourcing and DES by elaborating similarities and pointing out differences. Next, we consider opportunities and challenges when combining both approaches. We then propose a hybrid architecture that we are currently working on and finally give an overview on our future work.

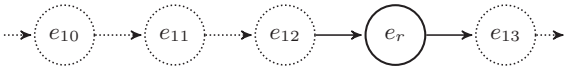
2. EVENT SOURCING & CQRS

Event sourcing is an architectural style that is used increasingly in large enterprise applications. As it is often combined with another concept—Command Query Responsibility Segregation (CQRS)—we will briefly introduce both. Then we consider typical distributed architectures that apply both styles together.

2.1 Event Sourcing

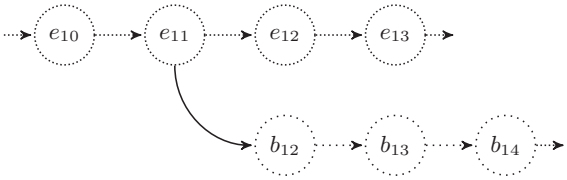
Fowler [6] summarizes event sourcing as "captur[ing] all changes to an application state as a sequence of events". Hence, event sourcing does not apply the traditional concept of state handling in sequential, imperative programming: to alter values by replacing them. Altering an entity's state in

(a) An unmodified event sourcing sequence with the five events $e_{10} \dots e_{14}$. Executing the sequence of all events yields the current application state.



(b) An event sourcing sequence $e_{10} \dots e_{13}$ with a retroactively injected event e_r between e_{12} and e_{13}

(c) An event sourcing sequence $e_{10} \dots e_{14}$ where event e_{12} has been retroactively removed from the log



(d) An event sourcing sequence $e_{10} \dots e_{14}$ where a branch $b_{12} \dots b_{14}$ has been created retroactively after event e_{11}

Figure 1: An unmodified event sourcing log that captures application state by storing sequences of events is shown in the first illustration. The other three illustrations demonstrate potential modifications of the given event sourcing log, retroactively changing application state or computing varying alternatives.

this way only captures the last state of the entity. Storing every single state-changing event instead might look like a considerable overhead at first. In fact, this style brings along several advantages as long as it is strictly adhered. Thanks to the event log, the system is not only aware of the final state of an entity, but also of it's entire history.

Using the event log, an application state can be rebuilt completely just by re-executing the stored sequence of events. It also allows us to recreate any prior state of the application by executing events until a certain point in the past. Interestingly, we can also alter the past by modifying events or event meta data, by removing events from the past or even by injecting new events into the past, as shown in Figure 1. A new computation of the modified event log yields the resulting alternate final state.

But not only the event log can be modified in event sourcing. Also the corresponding application code that handles events can be retroactively modified. Code changes include bug fixes, the introduction of new features or even additional code that executes time-dependent logic [6].

One could argue that such an architecture might be inefficient, because application states can only be obtained via the event logs. The architecture can be augmented using a secondary database that maintains application state in a more traditional way. In this case, events are not only appended to the event log, but also the state of the database is instantly updated. Note that this database can be volatile (e.g. an in-memory database), as its state can always be recomputed using the event log. Furthermore, snapshots of this secondary store can be persisted as snapshots that summarize the sequence of all preceding events. Snapshots are also necessary for fast re-executions after modifications of events or code. In this case, the last snapshot before the modified event is restored, and only the subsequent events are rerun.

2.2 Command Query Responsibility Segregation

In software engineering, Command-query Separation is a principle that promotes the differentiation of operations reading state and operations writing state. Command Query Responsibility Segregation (CQRS) is a concrete style following this principle [7]. A general advantage of this style is the reduced complexity of operations, as methods that tangle command-like operations (i.e. write) and query-like operations (i.e. read) are now discouraged. An architectural result is a separated model for commands and queries within the application.

When this separation is extended to the database, different stores may be used for readable data and for updates. In that case, both stores can be decoupled entirely and managed independently.

2.3 Applying Event Sourcing & CQRS in Distributed Event-driven Architectures

Event-driven architectures represent a popular solution for large-scale, distributed applications. This is mostly due to their non-monolithic design, their loose coupling of components, and their inherent asynchronous style, as illustrated by Hohpe [11]. Individual components of an event-driven architecture publish events and/or subscribe for events within the system. Event consumers process single events or sequences of events and may update internal state and publish new events. Some components apply a concept called (complex) event processing [13], where the component analyzes the stream of events inside the system and even matches patterns of events against complex rules.

Event-based systems can apply various styles for dealing with stateful applications. One of the options is the use of event sourcing and CQRS, as shown in Figure 2. Persistence and durability are guaranteed through event sourcing. All occurring domain events of the application are captured and logged into an event store. A secondary, read-optimized database maintains current the application state. In terms of CQRS, commands which represent state changes are added to the event log and also applied to the secondary database for updating. Queries do not interact with the event log. Instead, queries always use the secondary database for read operations.

The events may be propagated asynchronously to the secondary database. Such a weak coupling represents some kind of eventual consistency within the application: Not all commands that change states are instantly affecting the readable values. But after some time, values are also guar-

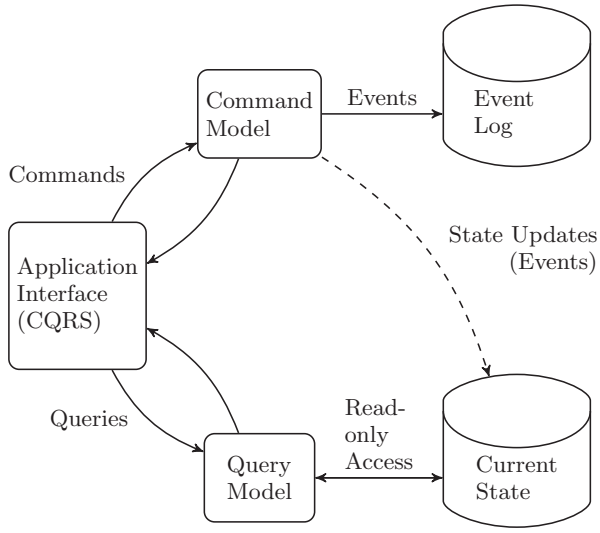


Figure 2: A section of an architecture applying event sourcing and CQRS. State changes are captured as events, added to an event store that logs all events, and forwarded to a secondary database maintaining the current application state.

anteed to be updated for reading requests. A weaker consistency is obviously not appropriate for all potential use cases. Many applications can actually handle it though, when eventually consistent behavior is already considered as part of the application design, and not only as an artifact of the database [10].

Another advantage of this design is the potential, but not mandatory separation of state handling. When strong consistency is required, the event store and the secondary database can be tightly coupled and guarantee visibility of changes. On the other hand, a decoupled design allows to fine-tune performance and scalability. For instance, applications challenged with highly concurrent read requests can be easily elastically adapted by scaling up the secondary stores.

There is an increasing number of libraries and frameworks that support the event sourcing and CQRS style in applications. For instance, the Eventsourced[1] library enhances the distributed computing framework Akka (Scala/Java) by applying event sourcing. Jdon[2] is a Java application framework for applications using event sourcing and CQRS. Microsoft has published a complete reference how to apply event sourcing and CQRS in their Azure cloud environment [1].

3. COMPARING EVENT SOURCING AND DES

Discrete event simulations (DES) represent the most common type of simulation in many domains, including networks, logistics, business processes, and various other domains with processes. As seen in the previous section, event sourcing represents a basic style for persistence in event-driven architectures, mainly used in distributed, large-scale enterprise applications.

[1]https://github.com/eligosource/eventsourced
[2]http://www.jdon.org/

In both approaches, events represent a fundamental principle: the progress of state over time. The current state of the execution is derived from all previous events. In a discrete event simulation, events are used to model real-world occurrences and are created and handled as part of the simulation program. In event-driven architectures, events are either created by external triggers (e.g. user requests) or by internal components, and handled by components of the application. Here, events are either predictable or unpredictable external stimuli the system reacts to or they are part of the internal application model itself.

While a discrete event simulation generates a deterministic simulation of potentially indeterministic real-world environments, event sourcing captures and stores events from the inherently non-deterministic real world in a deterministic and ordered manner. Both approaches yield a serialized timeline, where occurring events are ordered precisely using (logical) time stamps. For discrete event simulations, serializability is a key to validity and correctness. Only deterministic test runs are reproducible and enable repeatable executions. Ordering is also imperative for event sourcing, because otherwise the re-executions of an event log might yield diverging final states. Discrete event simulations as well as systems with event sourcing heavily benefit from parallelization methods. Such methods process multiple independent events in parallel, but still yield the same states as a serial execution does. This is also known as the causality constraint [8].

The architectural styles event souring and CQRS both have a strong relation to the software engineering approach *domain-driven design* (DDD). DDD encourages an explicit modeling of the application domain and the interaction of software developers and domain experts. Furthermore, implementation details are often abstracted from higher level domain concepts using domain specific languages (DSLs). This is very similar to the simulation domain, where the simulation engine, the simulation model and actual simulation program are also separated. In both cases, low-level implementation details are hidden from the application developer resp. simulation developer, who uses a DSL or other forms of abstractions. The underlying model is designed and implemented by experts that know both the domain and the architecture of the simulation/application. The actual application architecture or simulation engine is again developed differently with a focus on performance and other non-functional properties.

4. OPPORTUNITIES & CHALLENGES

We now focus on opportunities for simulation engines incorporating event sourcing concepts. Then we assess opportunities for a hybrid engine, running both applications and simulations.

4.1 Opportunities

We identified three main opportunities when applying event sourcing as a main concept of a simulation engine: (i) simplifications of interactive development and debugging, (ii) improvements in simulation analytics, and (iii) possibilities for distributed simulation architectures.

Current simulations environments provide limited support for developers when compared to other software development environments. Especially complex, large-scale simulations are difficult to debug and test due to very long exe-

cution times. At worst, simulation executions are abruptly aborted after hours of correct execution because of a single bug in the simulation code. There are guidelines how to trace errors and debug simulation models [12]. The most popular method is the creation of logging traces during execution and apply post-mortem trace analyses [5]. However, long executions times of large simulations render this approach cumbersome in practice. A direct and interactive development is not possible anymore, when the time between test runs, trace analysis, bug fixing and re-execution diverges.

We believe that event sourcing can help by making the internal event list of a simulator explicit to the developer through the event store. As we have seen in Section 2, event sourcing generally allows the modification of the event stream as well as modifications of the components handling the events. In doing so, simulation developers can explicitly analyze the captured events and are not required to create their own traces. Furthermore, simulation code can be modified in a paused simulation execution, or it can be retroactively modified and re-executed. When periodical snapshots of the event log have been created during the first run, a developer can jump to any event in the log, reestablish the corresponding simulation state and continue the execution without re-executing the prior simulation. Developers can also jump through the execution step-by-step (i.e. event-by-event on the event log and line-by-line within handling code). Stepping with introspection of values is an intuitive way of debugging sequential code. Depending on the event logging scheme, even an efficient reverse execution of events becomes possible: A developer restores a certain simulation state by selecting an event from the timeline, and then steps back in time by removing preceding events. The idea of reverse execution is not new for simulations. However, it has only been used as a rollback mechanism for optimistic parallelization efforts [2]. Reverse execution helps the developer to understand how unexpected simulation states have emerged.

Apart from simulation code modifications, the event log itself can also be altered. This allows the retroactive modification of events, of event meta data, or even of the simulation progress (branching). As the list of events in discrete event simulations are hidden from developer, this represents a novel approach towards interactive execution, debugging and parametrized execution. Alternative executions can be easily evaluated by retroactively injecting new events into the simulation event log and compute alternative outcomes on a branch. Single events caused by simulation programming errors can be corrected. A parametrized execution of a simulation also fits well into this model. A common timeline is computed, and branches with different event parameters split up the execution.

An explicit event log also pays off for debugging, when methods known from complex event processing (CEP) are used. Stepping through large simulation programs for debugging may be inefficient and cumbersome. A developer is rather looking for certain simulation states or certain state transitions. CEP engines provide query mechanisms for detecting complex event patterns. CEP query languages are similar to SQL, but additionally support temporal conditions. We believe that such CEP queries on event logs are a natural way of expressing breakpoints for interactive simulation debugging.

The explicit trace of all events captured in the event store allows the usage of a large set of concepts and tools for analyzing simulation results that original came from other event domains. The aforementioned CEP queries can also be used for simulation analysis and monitoring. Such an analysis component processes all occurring events on-line during execution of the simulation and yields additional results separate from the final state of the actual simulation. There is also active research on the explorative analysis of event traces with visual tools like Zinsight [4].

Major drivers for distributed simulations are the reduction of simulation execution time and the support of larger simulation models. Apart from traditional clusters, cloud environments are increasingly regarded for distributed simulations. While distributed simulation systems mainly focused on PDES concepts for the efficient execution of distributed simulations, the cloud has raised awareness for additional challenges of distributed simulation environments, such as fault-tolerance, heterogeneity and reliability [9]. Event sourcing does not suggest a new parallelization scheme for speed-up in distributed simulations. However, the event sourcing style provides some suggestions that can be mapped to simulation environments in the cloud. The separation of core application, event store, secondary database with current state and additional components allows easy distribution. The event store mainly requires large amounts of disk space, and efficient sequential access. The secondary database storing the current state is ideally stored in memory, hence should be placed on dedicated nodes with much RAM. The event store with event logging represents a native journaling mechanism. In case of a crash, a restart of the entire simulation can be avoided. Furthermore, the event store and secondary database can be replicated: the simulation can still continue to run even if one of the store instances fails.

4.2 Challenges

The crucial point of these considerations are causal relations between events and how they enable efficient re-executions. Otherwise, the fast re-execution of event logs is only possible by restoring snapshots and applying subsequent events. When event logs are modified (e.g. injection/modification/deletion of events), or when event handling code is modified, the system must be able to minimize the computational overhead to recompute the final state. We believe that this represents the major research challenge when applying the event sourcing style to discrete event simulations. The underlying problem is related to general causal relations in distributed computing [15] and vector clocks [14]. Recent advances in programming models for event-driven applications like EventWave [3] are promising to simplify the detection of event relations, but actual implications for event sourcing are still to be determined. We argue that additional causality information must be appended to the events in the simulation code and that explicit tagging by the developer as part of the programming model is more promising than purely automated mechanisms.

4.3 Towards a Hybrid Architecture

A particular advantage of a hybrid architecture for both simulations and applications arises when the actual application under simulation can again be implemented using event sourcing. For instance, business process engines or enterprise applications can be simulated using DES, but also

implemented with an architecture using event sourcing. A hybrid system combining both approaches allows not just the use of most of the application code for both simulation and real applications with a unified programming model. It also uses the same set of tools for development, debugging, monitoring, and analyses.

Most importantly, this architecture allows the seamless and instant transition from real-world, status quo application state to a simulated progression of the future. For the aforementioned domains, this is a real improvement when it comes to performance evaluations, predictions, and other kinds of analyses. Instead of exporting the current state of the application and importing it into an external simulation environment, a hybrid architecture can directly use the actual event sequence, create branches and execute simulations of the futures in the background. For example, a "branch" of a running business process system can be created while the systems continues to run. In the background, the branch can now be used as a starting point for a simulated execution with a behavioral model of users. At the same time, the original execution continues. At any time, comparisons of the actual event log and the logs of branched simulations can be performed.

5. FUTURE WORK & CONCLUSION

We have seen that discrete event simulations and applications using event sourcing are duals. A discrete event simulation imitates a real-world scenario over time by generating and handling events step-by-step. An application taking advantage of the event sourcing style uses a sequence of events to map the real world into a persistent, computational model.

Event sourcing has some interesting properties that we think are promising for discrete event simulations. By capturing and storing all events and not just the future events plus the current state, simulations runs can easily be paused, inspected, and stepped through both forwards and backwards. With the help of additional snapshots of the simulation state and the event log, already executed simulations can be efficiently restored at any point in time of the simulation. Most notably, event sourcing allows retroactive changes to the log of events as well as to the code that handle the events. Mapped to discrete event simulations, the event log does not only make the simulation navigable, it also enables very interactive development, debugging and parameterized branching.

We do not consider event sourcing as a silver bullet for discrete event simulations. Some issues, such as efficient parallel execution schemes for DES, are not addressed in particular. But we believe that obvious objections against an event sourcing style in simulations, namely large amounts of disk space for event logs and large amounts of memory for fast access to the current simulation state during execution, are not valid for modern cloud settings. Storage for event logs, memory optimized instances for current state, and computing-heavy instances for the simulation event execution are available on pay-per-use models. In return, simulation development, simulation debugging and simulation execution can become a more interactive, more explorative, and more comprehensible, thanks to the explicit event log.

Our current design of a hybrid architecture combines event sourcing with a traditional discrete event simulation mechanism. We already identified the main challenges for taking advantage of the new opportunities: The execution time of retroactively changed event histories must beat the execution time when starting a new simulation run from scratch. Hence, causal relations of events must be tracked and unnecessary re-computations should then be avoided. We plan to implement a prototype of our current design soon and evaluate the practicability of event sourcing for simulations in practice. According to our road map, we will then address the integration effort of complex event processing mechanisms such as queries and event stream visualizations.

6. REFERENCES

[1] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, M. Subramanian, and G. Young. *Exploring CQRS and Event Sourcing - A Journey Into High Scalability, Availability and Maintainability with Windows Azure.* Microsoft Developer Guidance, 2013.

[2] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9:224–253, 1999.

[3] W.-C. Chuang, B. Sang, S. Yoo, R. Gu, C. Killian, and M. Kulkarni. Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications. In *Proceedings of SOCC'13*, 2013.

[4] W. De Pauw and S. Heisig. Zinsight: a visual and analytic environment for exploring large event traces. In *Proceedings of SoftVis'10*, 2010.

[5] T. Dreibholz and E. P. Rathgeb. A powerful tool-chain for setup, distributed processing, analysis and debugging of omnet++ simulations. In *Proceedings of SimuTools'08*, 2008.

[6] M. Fowler. Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html, 2005.

[7] M. Fowler. CQRS. http://martinfowler.com/bliki/CQRS.html, 2011.

[8] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33:30–53, 1990.

[9] R. M. Fujimoto, A. W. Malik, and A. J. Park. Parallel and distributed simulation in the cloud. *SCS M&S Magazine*, 3, 2010.

[10] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[11] G. Hohpe. Programming without a call stack - event-driven architectures. Technical report, eaipatterns.com, 2006.

[12] D. Krahl. Debugging simulation models. In *Proceedings of the 37th conference on Winter simulation*, 2005.

[13] A. Margara and G. Cugola. Processing flows of information: from data stream to complex event processing. In *Proceedings of DEBS'11*, 2011.

[14] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, 1988.

[15] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7:149–174, 1994.