Conceptional problems of transaction-based modeling and its implementation in SimEvents 4.4

Lars Austermann, Peter Junglas, Jan Schmidt, Christian Tiekmann
PHWT Vechta/Diepholz/Oldenburg
peter@peter-junglas.de

Transaction-based modeling is a widely used graphical method for modeling discrete event systems, a recent implementation being SimEvents from Mathworks. Though it is applicable to a wide range of problems, it has its specific drawbacks. Some of them are connected to the basic abstractions of the method, others are related to the specific program used. By implementing a few different standard examples, we will show some of these shortcomings together with possible workarounds. This should be pointed out, when teaching this method to new users, but additionally has to be taken into account, when building a new transaction-based library or corresponding blockset.

1 Introduction

The modeling of discrete systems is a broad and difficult subject that has created a wealth of very different paradigms. Concentrating on graphical methods only, widely used techniques range from highly abstract ones such as Petri nets [1] and state graphs [2] to concrete material flow applications, e. g. PlantSimulation [3].

At a medium level of abstraction one finds process-based and transaction-based modeling [4], which both describe entities that are handled by fixed components. A widely used implementation of the former method is Arena [5], of the latter SimEvents [6]. The basic difference between them is the way how entities are transported: In process-based methods the components are the active parts and "seize" the entities, in transaction-based systems the entities play an active role and move automatically, until they are blocked. These two methods are used in many industrial applications, since they are sufficiently abstract to be universally applicable, but at the same time concrete enough to be comprehensible by users from very different disciplines.

But until now there is no well-established set of basic features and components for both methods. As a consequence users have to stick to a given simulation environment or even to a fixed version, if they don't want to reimplement and partially redesign their models. As the huge conceptional changes between the latest releases of SimEvents indicate [7], especially the definition of the transaction-based modeling approach presently seems to be unclear. Therefore one should reconsider the basic design of the method and try to answer questions such as the following: What are the shortcomings of current implementations? Which concepts or components are missing? How could a reasonable set of components be defined?

The aim of this paper is to provide first steps to answer these questions. To this end it examines a few standard example problems and their implementations in SimEvents 4.4. It shows the conceptual problems that have been encountered during the implementation and provides possible workarounds. It concludes with some proposals related to applying and teaching the transaction-based approach to modeling.

2 Example models

To study the range of applicability of the transaction-based approach, one can draw on a huge stock of practical examples. For the purpose of finding the weaknesses of the method, four models will be presented in the following, that clearly indicate the points that are of interest here. The first three come from Law's textbook [8], the last one from the Argesim benchmark C14 [9]:

timeshared Model of a time-shared computer, where several terminals send jobs of varying computing time demands, which are processed in time slices using a round-robin scheduler.

multiteller Model of a multiteller bank with several queues and jockeying, i.e. customers are allowed to change to a shorter queue.

jobshop Model of a factory with five workstations, where variable kinds of jobs are processed, which require different paths through the stations.

supplychain Model of a supply chain consisting of wholesalers, who order different products from distributors, which in turn order from several factories. The distributors use special strategies to comply with the demand.

The models are described in full detail in their references. The following section will concentrate on those parts of the implementations that are relevant for the discussion.

3 Problems implementing the examples

In the following some of the conceptual problems that have been encountered during implementation of the examples will be presented in detail. In addition it will be shown how to cope with them, mainly by introducing components that help to implement new abstractions.

3.1 Handling of concurrent events

The central element of many discrete event systems is a global event queue that contains all events in the proper order. But in a transaction-based model the events are defined locally by the individual blocks, so that the proper order of events that originate from different blocks at the same time instant is not always well-defined.

An example is shown in Fig. 1: A generator component creates entities with increasing id's, which will be routed afterwards to one of two outputs depending on whether the id is even or odd. For this purpose the `Get Attribute` block extracts the id of an

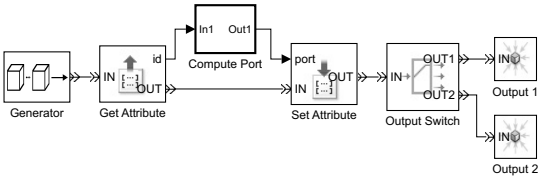

Figure 1: Example with concurrent events

entity, the `Compute Port` component uses this value to compute the corresponding output port number and the `Set Attribute` block sets the port number as an attribute of the entity.

Simulating the model leads to an error message stating that a race condition has been detected; it is not defined which of the two inputs of `Set Attribute` arrives first: the new entity or the new attribute value. If one ignores this message by changing it into a warning, the entities leave the `Output Switch` at the wrong port. But if one follows the recommendation given in the error message and inserts a *null server*, i.e. a `Server` component with service time $t_s = 0$, after the entity output of `Get Attribute`, the model works as planned.

This problem is inevitable when working with local components and is of course well known for a long time: Already the textual modeling language GPSS had a `BUFFER` command that reorders concurrent events [10].

The insertion of a null server to delay entities seems to be a simple solution, but it has a serious drawback: If its output is blocked, the server stores one entity. This has to be taken into account properly and complicates model design, as can be seen in the implementation of the CPU in the "timeshared" example (Fig. 2).

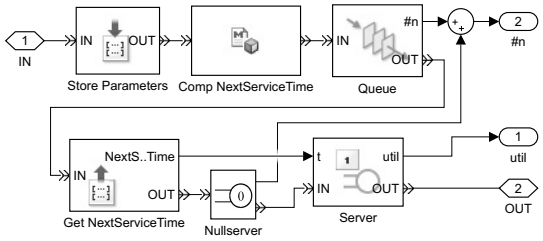

Figure 2: CPU component of the "timeshared" example

When jobs leave the queue to enter the server block that represents the CPU proper, their service time has to be fetched from an attribute, which makes the in-

sertion of a null server necessary. If one is interested in statistical data about the length of the queue, one has to add the additional job that may be stored in the null server. Therefore the average queue length that the queue component provides, is of no use, it has to be computed "manually" instead.

The workaround of using a null server to implement an "infinitesimal" delay is conceptually wrong, because the implicit storage that it adds, is not related to the problem it solves. This can even lead to more serious problems as will be seen in section 3.4.

3.2 Separation of entities from a queue

The "multiteller" example allows for the well known phenomenon of jockeying in a queue, i.e. customers at the end of a queue can switch to another shorter one. Modeling this behaviour with SimEvents turned out to be much more difficult than expected. The reason is that although an entity can leave a queue, when its waiting time exceeds a threshold (a behaviour known as *reneging*), there is no way to detach the last entity of a FIFO queue on arrival of an input signal (e. g. when another queue has become shorter).

To model such a *jockey queue* two very different schemes have been devised: In the *shuffle queue* (Fig. 3) the signal opens up a path from the exit to the beginning of the queue. All entities walk around the circle and get back into their old position, except for the last one, which leaves the block through the extra jockeying output. From here it is routed to the shorter queue.

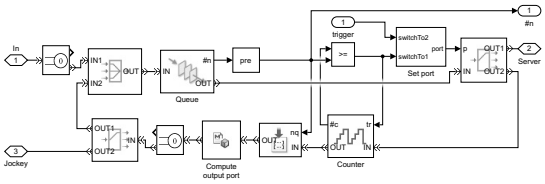

Figure 3: `Shufflequeue` component of the "multiteller" example

The *clone queue* (Fig. 4) creates duplicates of all incoming entities and routes them into a FIFO and a LIFO queue. When the queue exit is opened, an entity is taken from the FIFO queue, but when the jockey signal arrives, the LIFO queue is used. A bookkeeping device destroys clones, whose partner has already left the queue, when they appear at the end.

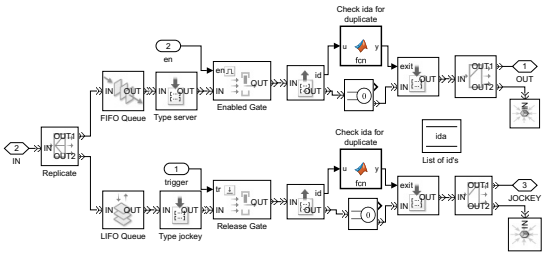

Figure 4: `Clonequeue` component of the "multiteller" example

Both schemes are quite complicated and lead to a lot of additional events. Though they worked at last, their implementations had to cope with a lot of difficult timing problems and appear to be cumbersome and error prone. What one needs instead, is a simple mechanism to remove a given entity from a queue, maybe similar to the concept of *user chains* in GPSS [10].

3.3 Storing entities

A basic ingredient of the "supplychain" example is a storage component that stores incoming entities denoting products of several types. On arrival of an order it emits the corresponding products at its output port. Scanning through the SimEvents library to find blocks that can store elements, one comes up with the queue, the server and the resource blocks, but none of them seems appropriate for the task at hand: A server is not well suited to deliver a certain type of product on demand, the resource pool provides a fixed amount of resources. And the queue block doesn't scale well with the number of different product types, since one needs one queue per type to emit a product entity of given type on request.

Instead of trying to use some of these blocks together with complicated gates and logic to coerce them into a non-fitting scheme, one can use a simple `1/z` block from Simulink's basic discrete library. It contains the inventory (Fig. 5), which is just a vector with the amounts of the different product types in the stock.

The `Storage` component (Fig. 6) registers incoming products in the inventory and destroys the corresponding entities. When an order arrives the inventory is reduced and the proper entities are recreated at the output port.

Of course this is only a trick, because the entities

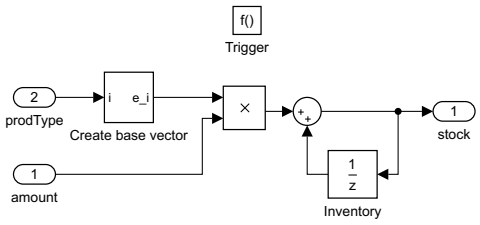

Figure 5: `Change Stock` component of the "supplychain" example

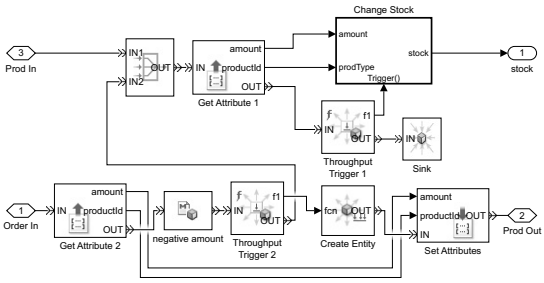

Figure 6: `Storage` component of the "supplychain" example

themselves are not stored at all. It worked for the example program, since the structure of the product entities is simple and always the same. What one really needs, is a more general component that actually stores the incoming entities and can release them on demand. The actual design of such a block is open to discussion and could be guided on example models and implementations in other environments. A simple first idea would be to use a special queue, where selected entities can move to the front of the line. Again, the old idea of "user chains" could come in handy here.

3.4 Time measurements across several blocks

In the "jobshop" example entities use different paths through workstations with associated queues. One is interested in statistics for the total waiting time of the entities over all those queues. To measure this value, one needs timers that can be paused after a queue and resumed before the next one to add up the single waiting times of passing entities.

Unfortunately SimEvents only provides simple timers that measure between two fixed points. Therefore one has to add up the waiting times of the individual queues inside the entity using an attribute (Fig. 7).

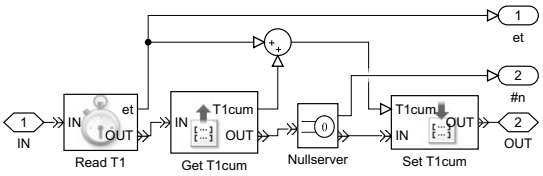

Figure 7: Accumulation of times in a `PauseCTimer` component

Based on this idea one can easily build components to start, pause, continue and read such accumulating timers and test them in simple models (Fig. 8).

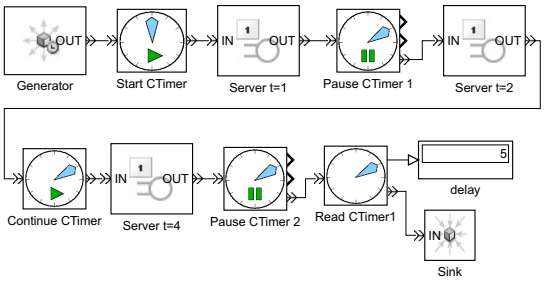

Figure 8: Simple test model for accumulating timers

But the problem is, they don't work if the accumulating block `PauseCTimer` sits between a queue and a server – which is exactly the most interesting configuration in general and especially in the "jobshop" example. The reason is of course the null server block that is necessary inside `PauseCTimer`: When the server is busy, the first entity in the preceding queue moves into the null server and waits there instead in the queue. The additional waiting time is not accounted for in the timer. This is a serious problem: One could measure the time that an entity stays in the null server, but to accumulate it, one needs another null server!

As always there is a workaround: Accumulation between a server and an (unlimited) queue is no problem, as the queue never blocks and the preceding null server doesn't store an entity. Therefore one can measure the time t_{Q+S} between entering the queue and leaving the server and the time t_S inside the server and accumulate the time $t_Q = t_{Q+S} - t_S$ after the server. But this is akward and only shows again that the basic design of a null server is seriously flawed.

3.5 Statistical analysis

An important aspect of discrete simulation is the gathering of statistical data, often in the form of a final report. In the "jobshop" example for instance, one is interested in the queue delay, queue length and server utilisation for the different workstations, as well as the total waiting time per job and per jobtype.

In a transaction-based environment there is no instance to collect such data but the blocks themselves. SimEvents provides mean values, utilisations and similar data for individual components, but no additional statistical tools. This is unfortunate, because due to the abundance of null servers one often has to combine individual values and can not use the statistical block data itself. The actual length of a queue for example has to be enlarged by the occasional inhabitant of a subsequent null server, as well as the corresponding waiting time. To compute mean values per entity or per time one has therefore to build own blocks, which admittedly can be done easily with standard Simulink methods.

Another challenge is the collection of statistics connected to entities, not blocks, such as the accumulated waiting times through several queues. The basic idea here is to store data in the entities themselves and collect them at the end. Fig. 9 shows how mean or maximal values can be computed with the help of simple `1/z` blocks that are hiding in the `Adder` and `Max` blocks.

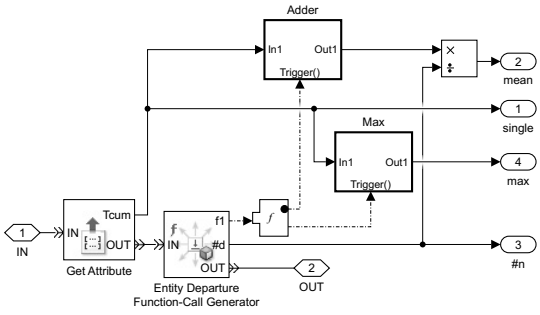

Figure 9: Component `EntityMeanMax` for accumulating entity data

Combining all these auxiliary blocks in a final statistics subsystem not only helps to unclutter the model, but brings together all statistical data. They now can easily be pipelined into a Matlab script that creates a proper report file, if such is desired.

4 Conclusions

The preceding analysis has unfold four different areas, where transaction-based modeling and its implementation in SimEvents show conceptional difficulties:

- timing of concurrent events;
- implementation of alternative queueing policies;
- storing and retrieving of entities;
- gathering and processing of statistical data.

Of these the first one is by far the most serious, and a generally working solution instead of the defective null server workaround is not available in SimEvents. This is especially annoying since already GPSS had a better solution with its `BUFFER` command.

All other problems could be solved by introducing appropriate subsystems using blocks available in SimEvents or the underlying Simulink. Adding such components to a user library, one can simplify the modeling of a wide range of applications. Though this may be sufficient for the practitioner in the industry, it is a real drawback of the transaction-based method: A corresponding library should provide the basic abstractions that are necessary to model all problems that the method adresses.

Considering the queueing and storing problems the GPSS construction of "user chains" seems to be a promising idea: Instead of adding ever more specialized components, it provides an underlying mechanism that may be able to cope with some of the difficulties presented above. To gain further insight into possible solutions, we propose to add a new benchmark to the Argesim suite [11] that requests the modeling of several jockeying queues and the collection of statistical data including the delay over several queues, similar to the multiteller example. To complicate matters one could ask for implementations with a large number of queues, which would adress the problem of "vectorising" components.

Mathworks has realized that SimEvents 4.4 has still basic problems and came up with a complete redesign of its SimEvents library with version 5. A very interesting feature is that the design is based on a unifying theoretical description [7]. Unfortunately, Mathwork has chosen a new design instead of relying on the well-known DEVS formalism [4]. Many basic aspects have

been changed with the new release, often by substituting graphical elements with Matlab code. As a consequence there is no simple migration path from the old to the new version. Whether this has lead to a satisfactory implementation, needs further investigation.

In any case this only adds to the central point made here: For the advancement of transaction-based modeling it is vital that it is based on a thorough theoretical analysis to reveal the fundamental abstractions and basic components that are necessary. The goal is to find stable designs, that don't change with every new tool or release, to get models that are better understandable, because they don't rely on tricky workarounds, and to reach high quality solutions, since they have a sound foundation. If we don't care for the basic concepts, we have to live with redesigning our models and rewriting our lectures every other year.

Acknowledgements

The second author (P. J.) likes to thank Thorsten Pawletta for introducing him to the rich history of discrete modeling tools.

References

[1] C. G. Cassandras, S. Lafortune. *Introduction to Discrete Event Systems.* Springer, New York, 2. ed. 2008.

[2] D. Harel. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 8, pp. 231-274, 1987.

[3] S. Bangsow. *Tecnomatix Plant Simulation: Modeling and Programming by Means of Examples.* Springer, Cham, 2015.

[4] B. P. Zeigler, H. Praehofer, T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, San Diego, 2nd ed. 2000.

[5] W. D. Kelton, R. P. Sadowski, N. B. Zupick. *Simulation with Arena.* McGraw-Hill, New York, 6. ed. 2015.

[6] The MathWorks. *SimEvents: Model and simulate discrete-event systems.* Online: `www.mathworks.com/products/simevents.html` (called 2017-01-30).

[7] W. Li, R. Mani, P. J. Mosterman. *Extensible discrete-event simulation framework in SimEvents.* Proc. 2016 Winter Simulation Conference, Arlington, pp. 943-954, 2016.

[8] A. M. Law. *Simulation Modeling and Analysis.* McGraw-Hill, New York, 5. ed. 2014.

[9] S. M. Tauböck. *C14 Supply Chain Management Definition.* Simulation News Europe, 32/33, pp. 42-43, 2001.

[10] T. J. Schriber, *On the application of user chains in GPSS.* Proc. 1973 Winter Simulation Conference, San Francisco, pp. 140-158, 1973.

[11] F. Breitenecker, S. Wassertheurer, N. Popper, G. Zauner. *Benchmarking of Simulation Systems — the Argesim Comparisons.* Proc. First Asia Int. Conference on Modelling & Simulation, Washington, pp. 568-573, 2007.