Speed up of Co-Simulation by a Heuristic Time Warp Mechanism

Christian Bartelt, Karina Rehfeldt and Stefan H. A. Wittek
Department of Software Systems Engineering, Clausthal University of Technolog, Clausthal-Zellerfeld, Germany

Keywords: Simulation, Time-Warp, Scheduling.

Abstract: Nowadays many engineered systems are modelled and simulated before their production. A common problem is that all modules and properties of complex systems cannot be modelled within only one simulation suite because they require different (proprietary) simulation software. This makes it desirable to be able to simulate a whole system simulation as cooperating simulation modules. To be efficient the communication between the modules has to be fast and must not be a bottleneck. In this paper we propose a theoretical concept to connect heterogeneous simulation modules. We utilize the mechanism of optimistic scheduling but expand it by using a heuristic to fast determine values. Our concept uses the rollback known from time warp mechanism. A module needs a certain amount of input data to process and when this data is not present at the given time of processing we use the heuristic to get all missing data. With these two enhancements we can limit the amount of rollbacks while speeding up the processing time of the whole system simulation.

1 INTRODUCTION AND RELATED RESEARCH

Cooperative processing of heterogeneous simulations is addressed by several approaches and software infrastructures (Bartelt et al., 2013). The Functional Mockup Interface (FMI) allows to orchestrate a number of slave simulation within a single master algorithm (MODELISAR, 2010). The definition of such algorithm is not part of the specification and has to be provided by an engineer who designs the co-simulation architecture and scheduling. The High-Level-Architecture (HLA) (IEEE, 2010) is a framework designed to integrate technically heterogeneous simulation modules. Its runtime infrastructure implements Jefferson's Time Warp protocols (Jefferson, 1985) as part of its time management (Fujimoto, 1998; Vardnega and Maziero, 2000). The Time Warp protocol aims to synchronize concurrent processes that communicating only by passing messages. The process receiving a message takes it as an input and may send output messages as a reaction on it. The received messages have to be processed in the order, in which they were sent, but may be delayed through the network connecting the processes. Time Warp allows the processes to handle the messages as they arrive. If a message with a time stamp is earlier than the last processed message arrives, a rollback is triggered. The process receiving this delayed message is reverted to the state at which the message should have been processed and sends so called anti messages to cancel the messages sent by this process based on this inconsistent sequence. This approached is called optimistic, since it assumes everything will work out well and later handles the occurring problem if this assumption is broken.

The approach, which is presented in this position paper, is based on the general Time Warp mechanism. However our presented approach proposes the usage of heuristics for an approximation of simulated values at runtime to avoid rollbacks. This allows to speed up co-simulations by parallel processing and optimistic scheduling. The protocol assumes that no messages will arrive out of order and provides rollbacks as a mechanism to fix the problems occurring if they do. We assume that the missing inputs will arrive at the module at a later time stamp, but that they will not change the output and provide mechanism to check, if they do, when they arrive. A rollback is only needed if such delayed input has a relevant impact. This reduces the amount of expensive rollbacks if the probability of changes is low. To allow the modules to start processing with missing inputs, these inputs are guessed using a predefined heuristic.

Bartelt C., Rehfeldt K. and Wittek S..
Speed up of Co-Simulation by a Heuristic Time Warp Mechanism.
DOI: 10.5220/0005107502670273
In *Proceedings of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2014), pages 267-273
ISBN: 978-989-758-038-3

There are also many alternative improvements of the Time Warp. Some researches focus on temporal uncertainty in distributed simulations (Beraldi and Nigro, 2000; Beraldi et al., 2002, 2002; Fujimoto, 1999; Quaglia and Beraldi, 2004), several other focus on more efficient rollbacks. This includes optimized storage mechanism for states (Prasad and Cao, 2003) and the definition of reverse operation to those operations changing the state of the process (Perumalla and Georgia, 1999) in order to avoid storing states at all. Other Works reduce the amount of rollbacks by adapting the optimism in the protocol. In (Ferscha, 1995) processes that have advanced further than the rest of the system are supposed to generate messages that will trigger rollbacks and slowed down. Another approach is to predict the timestamp of future messages, using the history of received messages (Srinivasan et al., 1995). Only messages received with a lower time stamp than this predicted time stamp are processed.

2 Optimistic Co-Simulation Scheduling

In our approach every simulation calculates a new value when it gets a message with a value from another simulation module. In most cases the simulation needs more than one input for complete and correct processing. It may not get all necessary data and missing data would have to be guessed. The processing with the guessed data would later be reversed when the correct data arrives. If the data is guessed in a very simple way, there is only a small chance, that the guessed values are close enough to the real ones. Therefore the calculation would have to be reversed. To increase the chances, we propose to use a heuristic provided by the modeller to predict missing values and avoid rollbacks.

To provide a mechanism to check predicted values, data is declared as unconfirmed when it is calculated with some estimated or with unconfirmed input data. Every unconfirmed data is marked with a condition. All unconfirmed data can still become a victim of a rollback. There exist two different times in the co-simulation. The calculation time is the time span a module needs for calculating a new value. The other time is the virtual simulation time, which is used for synchronization between the various modules. While the calculation time is represented by an actual time, the simulation time is a virtual time stamp. Due to the absence of synchronized clocks, this is widely used in distributed systems. All messages sent are marked with a time stamp. The time advances with every calculation in the following way: The new time stamp is the old time stamp plus the reaction time of the module.

In this section the proposed scheduling mechanism is explained based on a more formal component model for simulation infrastructures.
For a more intuitive explanation of the formalism, we introduce a simplified example of two robots in separated working spaces. The two robots are mounted onto skids (kinematics) and move along a circular rail. Each skid has its own control and is regulated by a controller. Figure 1 shows the structure of a corresponding co-simulation. Each robot has a kinematic, a controller and a control.

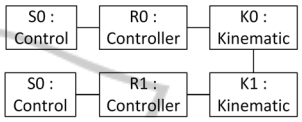

Figure 1: Schematic view of the co-simulation.

Both kinematics do not need to know the position of each other as long as they are not about to collide. The exact position only gets interesting when the two kinematics are close to each other.

2.1 Static View

The acting elements in this concept are modules. A module *MOD* is a tuple of eight sets of values (1). These sets represents the current state of the module and its behaviour.

$$MOD =_{def} CON \times HIS \times INPUT \times OUTPUT \times r \times lvt \times lgt \times f \quad (1)$$

$$CON =_{def} MOD^{*} \quad (2)$$

$$HIS =_{def} \{DATA \times STATE\}^{*} \quad (3)$$

$$INPUT =_{def} \{MSG\}^{*} \quad (4)$$

$$OUTPUT =_{def} DATA^{*} \quad (5)$$

$$DATA =_{def} TIME \times RESULT \times \{COND\}^{*} \quad (6)$$

$$COND =_{def} CONDR \cup CONDP \quad (7)$$

$$CONDR =_{def} MOD \times TIME \quad (8)$$

$$CONDPER =_{def} MOD \times TIME \times TIME \quad (9)$$

$$MSG =_{def} SIGN \times DATA \times MOD \quad (10)$$

$$SIGN = \{+,-\} \quad (11)$$

$$r \in Z \setminus \{0\} \quad (12)$$

$$lvt \in Z \setminus \{0\} \quad (13)$$

$$lgt \in Z \setminus \{0\} \quad (14)$$

$$f: MSG^{*} \rightarrow RESULT \quad (15)$$

The connection *CON* of a module is a set of all modules, which this is connected to (2). This may also include the module itself. In the history *HIS* are all past values together with a state saved (3). The state is needed to perform checks and rollbacks and should therefore save all relevant data to perform

them. What the state saves has to be decided by the designer of the module. In accordance to the time-warp concept a module contains two queues, an input queue *INPUT* and an output queue *OUTPUT (4-5)*.

The data needs a timestamp which indicates the simulation time for which the value was calculated and possible conditions *COND*. The conditions indicate that the calculation was based on heuristic data. Period conditions *CONDP* on output data indicates missing, respectively estimated input data. The timespan is between the referenced modules local guaranteed time (see below) and the timestamp of the output data. This is the timespan which is interesting for the calculated value. Referencing conditions *CONDR* on output data appear every time when input data with conditions, was used for a calculation (7-9).

Figure 2 gives an example for the creation of conditions. K0 is the calculating module and it gets two input data. On the calculated value comes up one referencing condition for R0, because the input data from R0 has a condition. And we get a new condition for K1, because the input data from K1 is not for timestamp 1 but 0 and therefore is a guessed value.

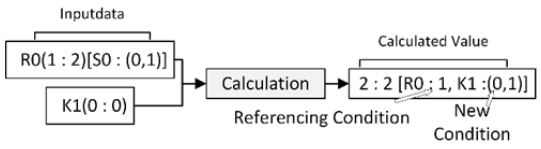

Figure 2: Example for conditions.

An Expansion of data is a message *MSG* (10). This is data together with the module which produced this data and a sign. The sign is used for deleting messages from the input queue in rollbacks (11). A negative message deletes its positive counterpart.

The module has got some time attributes, which are quite similar to the ones used by Jefferson. The time a simulated instance needs to 'react' in the real world is called r. This time period is not related to the actual calculation time, which can be a lot greater than r. It can also be called the step-width of the simulation.

A module has got a local virtual time (lvt) and a local guaranteed time (lgt). lvt is the current time of the simulation. lgt again, is the time of the last data without conditions.

Also important is the calculation function f (15). This is in fact the simulation itself which takes input data from the input queue and calculates new values.

Besides this every module has a heuristic for fast calculating or guessing values. To indicate that these values are 'guessed' they get a timestamp which is earlier than the real one. As prior stated this is later needed for determining the conditions.

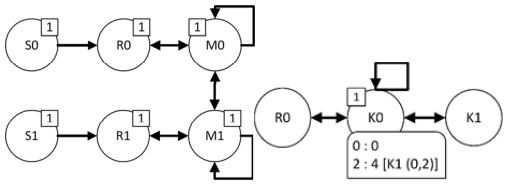

Figure 3: Example: Schematic view (l), module (r).

Figure 3 gives a schematic overview of the example and one module, respectively K0, one of the kinematics. The kinematic in this figure would be modelled in the following way (16).

$$
\begin{aligned}
&K0 = (C_{K0}, H_{K0}, I_{K0}, O_{K0}, 1, 2, 0, f_{K0})\\
&C_{K0} = \{R0, K1, K0\}\\
&H_{K0} = \{((2, 4, \{(K1, 0{,}2)\}), s_1)\}\\
&s_1 \in STATE\\
&I_{K0} = \begin{Bmatrix} (+, (0, 2, \emptyset), K1), \\ (+, (1, 5, \emptyset), R0), \\ (+, (2, 4, \{(K1, 0, 2)\}), K0) \end{Bmatrix}\\
&O_{K0} = \{(2, 4, \{K1, 0, 2\})\}
\end{aligned}
\tag{16}
$$

K0 is connected with R0, K1 and itself. In the history is a value 4 with timestamp 2 and a condition on K1 between 0 and 2. With the value the state of the module is saved. K0's input queue stores three values, one from each connected module. In the output queue is also stored the calculated value.

2.2 Dynamic View

Every module executes the simulate function itself. There is no need for a master component which has to control the execution order. This is due to the fact that every module just sends messages to others and changes only their own state.

2.2.1 Simulate

The simulate function transfers one state of a module into another (17). At first all pending data, respectively messages which arrived during the last simulation step, are inserted into the input queue with *putData*. When there is a message with a timestamp less than the module's lvt, then lvt is set back before the data's timestamp. But no rollback to an old state will be executed at this step. Before the rollback takes place, a check is done, which will determine whether a rollback is necessary or not.

$$
\begin{aligned}
&simulate : MOD \rightarrow MOD \\
&simulate(m) = m' \\
&m_0 = putData\big(m, getPendingData(m)\big) \\
&t = nextTimestamp(m_0) \\
&D = nextData(m_0, t) \\
m' &= \begin{cases} calculate(m_0, t, D), \\ if\ \nexists d\big(d \in History_{m_0} \wedge Time_d = t + r_{m_0}\big) \\ checkExisting(m_0, t, D), \\ else. \end{cases}
\end{aligned} \tag{17}
$$

After inserting the data in the input queue, the next timestamp for calculation is determined.

With *nextData* all required data for the prior determined timestamp is collected.

When there is already a calculated data for timestamp t+r in the history, this one has to be checked. This will occur if lvt was set back while processing incoming messages. Furthermore this means that a connected module now has calculated new data which was prior heuristically calculated. So the module needs to check if the 'real' data changes anything for its calculation. A rollback is performed if the former calculated data was too inaccurate. Otherwise the data may change its conditions but a rollback is averted.

2.2.2 Receiving and Sending Data

Every time a module inserts a new value in its history or changes a condition it sends a message to all connected modules about this. For this it uses the *send* function for every module.

$$send : MOD \times MSG \rightarrow MOD \tag{18}$$

$$putData: MOD \rightarrow MOD \tag{19}$$

Send inserts the message into a temporary buffer of the module to avoid unwished side effects (18). It fulfils our condition, that the critical state of a module isn't changed by another one.

The temporarily saved messages are inserted into the input queue with *putData* (19). It also adjusts the time if required. The module can only process data which has a timestamp bigger than its lvt. If a message with a timestamp earlier than module's lvt arrives, lvt is set back to an even earlier time. The order in which the messages are taken out of the buffer and inserted into the input queue does not matter. The execution order will stay the same, because lvt is always set to the earliest timestamp of all processed messages

Consider now our example again. Especially we are looking at K1. The input queue of K1 is assumed as in (20) and lvt of K1 is three.

$$
\begin{aligned}
I_{K1} &= \begin{Bmatrix} (+, (1, 3, \emptyset), R1), \\ (+, (2, 4, \{K1, 0, 2\}), K0) \end{Bmatrix} \\
lvt_{K1} &= 3
\end{aligned} \tag{20}
$$

We assume that K1 got a message from its controller R1 and from K0. In the temporary buffer are therefore two messages.

$$buffer = \{(+, (1, 1, \emptyset), K0), (-, (1, 3, \emptyset), R1)\} \tag{21}$$

putData will insert the messages in the input queue and because one is less than two (lvt-r) it will set the local virtual time of K1 to one. The second message is a negative message. It will delete its positive counterpart in the input queue. So after inserting it the input queue has gotten smaller.

$$
\begin{aligned}
I_{K1} &= \begin{Bmatrix} (+, (1, 1, \emptyset), K0), \\ (+, (2, 4, \{K1, 0, 2\}), K0) \end{Bmatrix} \\
lvt_{K1} &= 1
\end{aligned} \tag{22}
$$

2.2.3 Calculating Data

$$nextTimestamp : MOD \rightarrow TIME \tag{23}$$

$$nextData: MOD \times TIME \rightarrow MSG^* \tag{24}$$

$$calculate : MOD \times TIME \times MSG^* \rightarrow MOD \tag{25}$$

$$condRef : MOD \times DATA \rightarrow CONDR \tag{26}$$

$$condPer : TIME \times MOD \times DATA \rightarrow CONDP \tag{27}$$

$$
\begin{aligned}
&nextvalue : MOD \times MSG^* \rightarrow DATA \\
&(m, M) \mapsto (t, v, C) \\
&t = max(Time_{n.Data}, \forall n \in M) + r_m \\
&v = f_m(M) \\
&C = \bigcup_{i \in M} condPer(max(Time_{n.Data}, \forall n \in M), Modul_i, Data_i) \\
&\qquad \cup condRef(Modul_i, Data_i)
\end{aligned} \tag{28}
$$

To calculate new data one has to know at first for which timestamp. To get this the smallest timestamp in the input queue bigger than lvt-r is searched with nextTimestamp (23).

Afterwards, nextData gets this timestamp and searches for all data in the input queue of the module which has the same timestamp (24). Now some data might still be missing. All modules which had not provided data with the searched timestamp are asked for a heuristically calculated value. *nextData* returns a set of all needed input values.

The gathered data is now given to *calculate* (25). At first *nextvalue* is called by *calculate* (28). It calculates a new result value with the f-function of

the module. It uses the condition-functions to determine all necessary conditions. After that *calculate* sends the data to all connected modules and saves the data in the module's own history and output queue. At last *calculate* increases the local virtual time. It takes the timestamp of the input data (which is bigger than lvt-r) and adds r to it. Therefore, time advances over the simulation steps.

The two types of conditions are calculated by (26) and (27). *CONDP* are created if the time of the input data is smaller than the given time. A *CONDR* is created every time the input has conditions.

Again we will show the functions on our example. K0 might look like in (29) before.

$$\begin{aligned} &K0 = (C_{K0}, H_{K0}, I_{K0}, O_{K0}, 1, 1, 1, f_{K0}) \\ &C_{K0} = \{R0, K1\} \\ &H_{K0} = \{((1, 1, \emptyset), s_1)\} \\ &I_{K0} = \{(+, (1,2, \{(S0,0,1)\}), R0)\} \\ &O_{K0} = \{\} \end{aligned} \tag{29}$$

Then the functions (30-32) are called sequentially. Which leads to the state of K0 as seen in (33)

$$nextTimestamp(K0) = 1 \tag{30}$$

$$\begin{aligned} nextData(K0,1) = \{&(+, (1,2, \{(S0,0,1)\}), R0)\} \\ &\cup bestvalues(1, K1) \end{aligned} \tag{31}$$

$$calculate(K0,1, nextData(K0,1)) \tag{32}$$

$$\begin{aligned} &K0 = (C_{K0}, H_{K0}, I_{K0}, O_{K0}, 1, 2, 1, f_{K0}) \\ &H_{K0} = \begin{Bmatrix} ((1, 1, \emptyset), s_1) \\ ((2, 2, \{(R0, 1), (K1, 0, 1)\}), s_2) \end{Bmatrix} \\ &I_{K0} = \{(+, (1,2, \{(S0,0,1)\}), R0)\} \\ &O_{K0} = \{(2, 2, \{(R0, 1), (K1, 0, 1)\})\} \end{aligned} \tag{33}$$

The newly calculated value was inserted in the history and output queue and sent to the connected modules. The input queue was not changed because the data might be needed for a later calculation.

2.2.4 Check Existing Data

If there is already data in the history for a given timestamp, this data is checked with *checkExisting* whether it should be updated (34). It may trigger a rollback but with good heuristics and light dependencies between the modules this can be prevented. We make the assumption that some light changes in the input data will not cause an effect on calculations and use this to speed up a whole simulation.

$$\begin{aligned} &checkExisting: MOD \times TIME \times MSG^* \rightarrow MOD \\ &checkExisting(m, t, D) = m' \\ &DATA \ni d = (d \in History_m \mid Time_d\ t + r_m) \end{aligned} \tag{34}$$

$$m' = \begin{cases} updateData(m_0, d, D), \\ \quad if\ conditions\ change \\ calculate(rollback(m_0, t), t, D), \\ \quad if\ d\ incorrect. \end{cases}$$

$$updateData: MOD \times DATA \times MSG \rightarrow MOD \tag{35}$$

UpdateData updates the conditions of an existing data for new input but will not change the value (35). What happens to the conditions of the data depends on the module. The old data is deleted from history and output queue and the connected modules are informed about the update. This should not be confused with a rollback. In the connected module the message will just trigger an update, not a rollback.

For our example this could mean that when K0 gets a new value from K1 it may not have to do a new calculation. Assume the former calculated position of K0 was 10 and there was no collision. The new position of K1 now is 20. Clearly this has no effect on the old calculation, so instead of doing a rollback only a condition has to be deleted.

In our example K0 is in the state of (36). After executing *checkExisting* it is in the state like in (38).

$$\begin{aligned} &K0 = (C_{K0}, H_{K0}, I_{K0}, O_{K0}, 1, 2, 1, f_{K0}) \\ &C_{K0} = \{R0, K1\} \\ &H_{K0} = \begin{Bmatrix} ((1, 1, \emptyset), s_1) \\ ((2, 2, \{(K1, 0, 1)\}), s_2) \end{Bmatrix} \\ &I_{K0} = \begin{Bmatrix} (+, (1,2, \emptyset), R0) \\ (+, (1, 0, \emptyset), K1) \end{Bmatrix} \\ &O_{K0} = \{(2, 2, \{(K1, 0, 1)\})\} \end{aligned} \tag{36}$$

$$checkExisting(K0,1, nextData(K0,1)) \tag{37}$$

$$\begin{aligned} &nextData(K0,1) \\ &= \{(+, (1, 2, \emptyset), R0), (+, (1, 0, \emptyset), K1)\} \end{aligned}$$

$$\begin{aligned} &H_{K0} = \begin{Bmatrix} ((1, 1, \emptyset), s_1) \\ ((2, 2, \emptyset), s_2) \end{Bmatrix} \\ &O_{K0} = \{(2, 2, \emptyset)\} \end{aligned} \tag{38}$$

We assume that *checkExisting* will call *updateData* because the new value of K1 just changes the conditions of (2, 2). The state of K0 is then changed by *updateData*. After that it looks like in (38).

While the execution of *updateData* the old value $(2, 2, \{(K1, 0, 1)\})$ from the outputqueue is sent as a negative message and afterwards deleted.

2.2.5 Rollback

Rollback deletes all data before the given timestamp (39). Every message which is deleted in the output queue is sent to all connected modules as a negative

one. This will delete their positive counterparts in the input queue of connected modules.

$$rollback{:}MOD \times TIME \rightarrow MOD \quad (39)$$

3 CONCLUSION, LIMITATIONS AND OUTLOOK

The efficient processing of coupled heterogeneous simulations of engineering products is a serious challenge. In many (co-)simulation infrastructures, all connected simulation modules wait for the slowest part though parallel processing. To overcome this problem, a theoretic approach to schedule efficiently the parallel processing of connected simulation modules is presented based on a formal component model. The optimization potential of this approach depends on a suitable heuristic for the interaction behaviour of connected simulation modules. To evaluate the approach, the component model was implemented and tested by an exemplary simulation scenario.

Our approach is subject to some assumptions which limit the spectrum of applicable co-simulation environments: A central aspect of our approach is the use of a heuristic to provide fast the result values of slow modules. To design such heuristics, domain knowledge is needed, so we rely on the experts modelling the module to provide it. The quality of this heuristic has high impact on the occurrence of rollbacks in or approach and so on its potential to speed up the simulation. Additionally the coupling structure of co-simulated modules determines the possible speedup. Scenarios in which the modules form dents webs around a single slow module tend to produce more rollbacks. This is due to the fact that the heuristically produced data is used at many distinct modules. The probability that one of these modules will produce a different output using the accurate data simply adds up.

In further work more complex case studies for simulation of whole tool machines are currently implemented. Additionally further research on more sophisticated heuristics for the prediction of connected simulation behaviour is planned.

REFERENCES

Bartelt, C., Böss, V., Brünning, J., Denkena, B., Rausch, A., Tatou, J.P., 2013. A Software Architecture to Synchronize Interactivity of Concurrent Simulations in Systems Engineering, in: In Proceedings of the 20th ISPE International Conference on Concurrent Engineering.

Beraldi, R., Nigro, L., 2000. Exploiting Temporal Uncertainty in Time Warp Simulations, in: Proceedings of the Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications, DS-RT '00. IEEE Computer Society, Washington, DC, USA, p. 39–.

Beraldi, R., Nigro, L., Orlando, A., Pupo, F., 2002. Temporal Uncertainty Time Warp: An Agent-Based Implementation, in: Proceedings of the 35th Annual Simulation Symposium, SS '02. IEEE Computer Society, Washington, DC, USA, p. 72–.

Ferscha, A., 1995. Probabilistic Adaptive Direct Optimism Control in Time Warp, in: In Proceedings of the 9th Workshop on Parallel and Distributed Simulation. pp. 120–129.

Fujimoto, R.M., 1998. Time Management in the High Level Architecture. Simulation 71, 388–400.

Fujimoto, R.M., 1999. Exploiting temporal uncertainty in parallel and distributed simulations, in: Thirteenth Workshop on Parallel and Distributed Simulation, 1999. Proceedings. Presented at the Thirteenth Workshop on Parallel and Distributed Simulation, 1999. Proceedings, pp. 46–53. doi:10.1109/PADS.1999.766160

IEEE, 2010. std 1516-2010, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules. The Institute of Electrical and Electronic Engineers.

Jefferson, D.R., 1985. Virtual time. ACM Trans. Program. Lang. Syst. 7, 404–425.

MODELISAR consortium, 2010. Functional Mock-up Interface for Co-Simulation v1.0 [WWW Document]. URL https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_CoSimulation_v1.0.pdf (accessed 4.30.14).

Perumalla, K.S., Georgia, R.M.F., 1999. Efficient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul. 9, 126–135.

Prasad, S.K., Cao, Z., 2003. Parallel Distributed Simulation and Modeling Methods: SyncSim: A Synchronous Simple Optimistic Simulation Technique Based on a Global Parallel Heap Event Queue, in: Proceedings of the 35th Conference on Winter Simulation: Driving Innovation, WSC '03. Winter Simulation Conference, New Orleans, Louisiana, pp. 872–880.

Quaglia, F., Beraldi, R., 2004. Space uncertain simulation events: some concepts and an application to optimistic synchronization, in: 18th Workshop on Parallel and Distributed Simulation, 2004. PADS 2004. Presented at the 18th Workshop on Parallel and Distributed Simulation, 2004. PADS 2004, pp. 181–188. doi:10.1109/PADS.2004.1301299

Srinivasan, S., Srinivasan, S., Jr, Reynolds, P.F., Reynolds, P.F., 1995. NPSI Adaptive Synchronization

Algorithms for PDES, in: In 1995 Winter Simulation Proceedings. pp. 658–665.

Vardnega, F., Maziero, C., 2000. A Generic Rollback Manager for Optimistic HLA Simulations, in: Proceedings of the 4th IEEE International Workshop on Distributed Simulation and Real-Time Applications. pp. 79–85.