I. Troch, F. Breitenecker, eds. ISBN 978-3-901608-35-3

EFFICIENT SIMULATION OF HYBRID CONTROL SYSTEMS IN MODELICA/DYMOLA

L. Liu[1], G. Frey[2]
[1] University of Kaiserslautern, Kaiserslautern, Germany, [2] DFKI - German Research Center for Artificial Intelligence, Kaiserslautern, Germany

Corresponding Author: L. Liu, University of Kaiserslautern, Department of Electrical and Computer Engineering, Erwin-Schrödinger-Str. 12, 67663 Kaiserslautern, Germany, liuliu@eit.uni-kl.de

Abstract. Though systems with mixed discrete/continuous behaviors can be handled by proper hybrid simulation tools, the efficiency of simulation is often not satisfactory. The performance of simulation is affected by various system properties such as the frequency of events, the number of continuous state variables, etc. In an ordinary modeling and simulation approach, the hybrid system is considered as a whole thus every local event has global effects. I.e. An event triggered by any element of the system causes the numerical solver to detect the event and recalculate all state variables. This works well if the discrete event logic couples tightly with continuous behavior evolution. A tight coupling means that the discrete event sub-system governs the whole system and all its events necessarily trigger the switch of equation system or stop/restart of the numerical solver. However, in case of a loose coupling, i.e. not all the events from the discrete event sub-system are relevant to the continuous sub-system, this approach is excessively expensive. This paper gives a general approach to achieve more efficient simulations of hybrid systems with loose coupling, especially for stiff systems such as the Networked Control/Automation Systems (NCS/NAS) in which different system components have strongly distinguished temporal characteristics. Advantages of this approach are demonstrated using the simulation tool Modelica/Dymola.

1 Introduction

Hybrid control systems generally involve both continuous and discrete dynamic behavior. In the context of digital control systems (Figure 1), the discrete sub-system representing the digital controller and sampling devices governs the continuous sub-system representing the plant under control by specifying discrete event switching logic. On the other hand, the continuous sub-system operates according to newly assigned states.

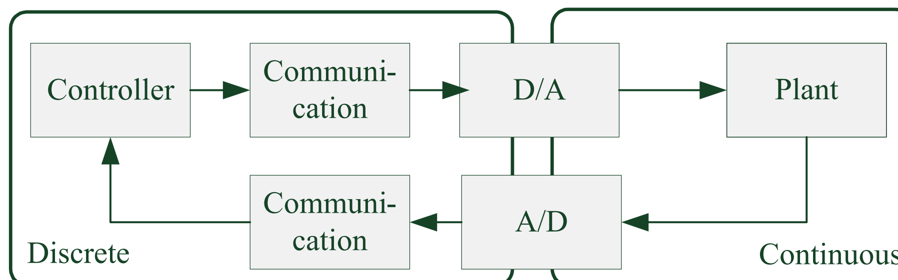


Figure 1. Hybrid control system structure

Recently, there is increasing interest in Networked Control/Automation Systems (NCS/NAS), i.e. systems with communication network and decentralized controller units. Diverse embedded devices are interconnected using a network. The continuous sensor values are sampled and converted by the Analog-to-Digital Converter (A/D), the resulting information is then transmitted to controller through network. Based on the sensor values, the given control laws generate appropriate controller output, it is then transmitted to the Digital-to-Analog Converter (D/A) and finally to the actuator to control the plant (sensors and actuators are considered part of the plant).

The use of an object oriented modeling paradigm simplifies the modeling of complicated hybrid systems. Consequently, simulation is commonly utilized for comprehensive analysis of hybrid systems. Prerequisite of a hybrid simulator is the capability to detect and locate events as well as solve for consistent initial/restart values of state variables. Existing simulation tools may handle this issue using different numerical solvers with event handling capability. However, the high computational cost remains as a crucial problem among them.

For the analysis of NCS/NAS, the Modelica NC-Library has been developed using object oriented modeling language Modelica [1]. The library supports the analysis of NCS/NAS together with the plant models in a closed loop. Main advantage of this approach is that the overall system analysis can be executed in a single simulation environment, e.g. Dymola. The NC-Library provides device libraries including detailed models of controller units, communication network, etc. However, the detailed models introduce a considerable number of events such as communication events for indicating the communication states and processing events for exhibiting the controller states, etc. Consequently, the simulation speed is often not satisfactory. In the analysis of NCS/NAS,

events are relevant to correct temporal and functional behavior of the system and thus cannot be eliminated or even partially ignored. Obviously, reducing the number of events is not an option for accelerating the simulation. Therefore, a new method is required.

The paper presents an approach to accelerate the simulation of hybrid control systems. The method is based on classification of events and separation of event domains. It provides efficient simulation while holding the accuracy of simulation results. The approach is implemented and tested using Modelica/Dymola. In the following discussion, the continuous system dynamics are supposed to be described using Differential and Algebraic Equation systems (DAEs) and there is an appropriate tool (e.g. Dymola) to solve them. The details about DAE solver are not concerned in this paper.

The paper is organized as follows: Chapter 2 describes the fundamentals of hybrid simulation concerning simulation performance. In Chapter 3, potential approaches for improving the simulation performance are discussed. Chapter 4 explains the principles of separated simulation including the synchronization scheme and the classification of events. In Chapter 5, the implementation of separated simulation in Modelica is introduced. Followed by a comparison of simulation performance using integrated simulation and separated simulation in Chapter 6. Finally, conclusions are given in Chapter 7.

2 Hybrid simulation fundamentals

It is well know that numerical integration methods require the model equations to remain continuous and differentiable. But in the context of hybrid simulation, this assumption usually not holds because of event switching. Take the following function as an example:

$$\dot{x} = \begin{cases} -x & \text{if } y < 0 \\ x & \text{if } y \geq 0 \end{cases} \tag{1}$$

In the given case, it handles about a state event. The value of y is a guard for switching between different branches. Assume that y is a function of time described by $y = g(t)$. The hybrid simulator tries to detect the event by checking the value of y. The numerical integrator proceeds further through variable time steps until the value of y crosses the non-differentiable point $y = 0$. Then it can be determined that in the last simulation step $(t_{s-1},\ t_s]$ the event has occurred. Since the simulation step widths are selected without consideration of the dynamic of the guard function, no accurate time of occurrence can be given. Based on a predefined tolerant bound, now a localization procedure takes place. In the localization, root finding algorithms are used to find the approximate root of the guard function under certain tolerance. As a result, the event is localized and time t_e is found as the time of occurrence. The integration is halted at this time point, the new branch is selected, if necessary, consistent restart values are calculated, and the integration is started again [2], [3].

In case of time events, the times of occurrences are explicitly specified. Thus the detection and localization of events are not required. The rest is handled in the same way as state events. Obviously, time events can be treated more efficiently. But in hybrid control systems, it is rare to define time events strictly, except for the sampling events of sensors and actuators. Most other events have to be handled as state events.

Although the basic principle sounds simple, the computational cost behind is not negligible. To define computational cost, two important marks are selected: One is the CPU time needed for a certain simulation time; the other one is the amount of memory needed for storing simulation results. Both are affected by a number of factors from the models themselves and the simulation settings including the frequency of events, the number of state variables, the chosen integration algorithm, the length of the output interval, and so on. Some of the factors also interact with each other. For example, the interval length of output values has to be selected accordingly with the frequency of events as well as the temporal characteristics of state variables to ensure usable results.

A benchmark test has been carried out to show how the factors affect the simulation performance. The benchmark model is given in Figure 2. In this model, a switch condition is defined by a sine function $x = \sin(2\pi f \cdot time)$. The frequency is defined as a parameter so that the frequency of state events can be adjusted. The number of state variables and auxiliary variables can also be modified. By the occurrence of discontinuous point $|x| = 0.99$, derivations of state variables as well as the value of auxiliary variables are switched between two different trigonometric functions. This model examines the detection and localization of events as well as recalculation of consistent values of state variables. Five tests have been carried out and results are shown in Table 1. The parameters of the models are listed in the first three light shadowed rows. The simulation settings are listed in the two middle rows while the simulation performances are shown in the last four rows. The simulations were performed in Dymola 6.1 on a 3.0 GHz Pentium 4 hyper threading processor with 2 GB of RAM running Windows XP. Each test simulates the model for 1 second. Symbol '*' means the simulation does not yield acceptable results because of inappropriate simulation setting. Two integration algorithms are tested. Generally, Lsodar is more effective than Dassl in this benchmark test. The difference may come from the distinct implementations of event detection/localization, random selection of variable integration step size, root finding algorithm, etc. Detailed information about these two solvers can be found in [5], [6], [7].

```
model EventStates
  import Modelica.Constants.pi;
  constant Real frequency=1000;
  parameter Integer numberOfStateVariables=100;
  parameter Integer numberOfAuxiliaryVar=100;
  Real y[numberOfStateVariables];
  Real z[numberOfAuxiliaryVar];
  Real x;
equation
x=sin(2*pi*frequency*time);//switch condition
  // two branches for state variables
  if abs(x)>=0.99 then
    for i in 1:numberOfStateVariables loop
    der(y[i])= sin(2*pi*frequency*time);
    z[i]=sin(time);
    end for;
  else
    for i in 1:numberOfStateVariables loop
    der(y[i])= cos(2*pi*frequency*time);
    z[i]=cos(time);
    end for;
  end if;
end EventStates;
```

Figure 2. Benchmark model

	no.1	no.2	no.3	no.4	no.5
Num. state variables	100	100	200	100	100
Num.auxiliary variable	100	100	200	100	100
Frequency	1000	1000	1000	2000	1000
Algorithm	Lsodar	Lsodar	Lsodar	Lsodar	Dassl
Output interval (s)	1e-4	1e-5	1e-5	1e-5	1e-5
Detected state events	1058*	4000	4000	8000	1000
CPU time (s)	2.05*	10.2	21.5	13.8	99
Num. result points	12117*	108001	108001	116001	108001
Size. result file (MB)	9*	130	260	140	130

Table 1. Results from benchmark test

The number of state events is determined by the parameter *frequency*. For a frequency of 1000 Hz, there are 4000 events in 1 second simulation time. Test no.1 failed to detect the state events completely because of the inappropriate output interval. The output interval indicates the time gap in which the values of variables are compulsively updated. If the output interval is not sufficiently small, the value of x is not updated in time thus the guard condition is crossed without activation. In hybrid simulation, the values of state variables are stored not only at each continuous integration step but also at the occurrence of events. Moreover, by event time point t_e, a sufficiently small time step ε is given so that the condition of event is false at time $t_e - \varepsilon$ and is true at time $t_e + \varepsilon$. Consequently, results are stored twice at each event. Finally, the number of result points n_r is a function of the number of events n_e in the given simulation time t_s as well as the output interval τ and is given by

$$n_r = 2n_e + t_s / \tau + 1 \tag{2}$$

Back to the evaluation of computational cost, the results show that the memory consumption m_c is a function of the number of state variables n_s, the number of auxiliary variables n_a, and the number of result points n_r. Depending on the setting of simulation, not only the state variables themselves but also their derivations are stored. Thus the total amount of time varying variables equals to $2n_s + n_a$. Along with the proportionality coefficient k which describes the unit memory consumption for a variable with certain precision, the memory consumption of simulation is defined by

$$m_c = k \cdot (2n_s + n_a) \cdot n_r \tag{3}$$

The case for CPU time is more complicated. Generally, three different time constants are taken into consideration. Each of them signifies the mean value of processing time required for corresponding procedures concerning either each state variable or each time varying variable. t_{sc} denotes the CPU time needed for one successful continuous integration step. t_{se} represents the CPU time of detection/localization/recalculation for each state event. t_{st} indicates the CPU time for storing one variable in memory and disk. The total CPU time t_c for simulation is then approximately given by

$$t_c = (t_s / \tau + 1) \cdot t_{sc} \cdot n_s + n_e \cdot t_{se} \cdot n_s + n_r \cdot t_{st} \cdot (2n_s + n_a) \tag{4}$$

Generally speaking, the computational cost is proportional to the number of time varying variables and the number of events. Besides, the required simulation accuracy also plays an important role.

3 Methods for improving the simulation performance

It has been observed from Chapter 2 that a hybrid simulation is excessively time-consuming in case of frequent events and a large equation system. The most straightforward and reasonable method for improving the simulation performance is to increase the computational power. However under most circumstances, the computational power of desktop computers is limited by both hardware and software. Thus for complex systems a special technique called parallel/distributed simulation has been developed. It allows a simulation program to be executed on parallel/distributed computer systems, namely systems composed of multiple interconnected computers [4]. However, this technique requires special hardware structures and software support not readily available under normal laboratory conditions. Yet other solutions have to be investigated.

Under certain restrictions of computational power, the other way out for accelerating simulation speed is to reduce the complexity of simulation. Obviously, the complexity of simulation is directly decided by the complex-

ity of models and the required accuracy. From the analysis in Chapter 2, a qualitative definition of the complexity of simulation C_s in a given simulation time t_s can be approximately described by

$$C_s = g(n_e, n_s, n_a, \tau) \quad (5)$$

However, the correctness of the simulation result highly depends on the model's level of accuracy. Generally speaking, well defined models consist of a minimum set of necessary variables and events which can not be further reduced. In other words, it is not acceptable to improve the simulation speed at the cost of accuracy.

Analysis from (3) and (4) shows that the computational costs are mainly decided by the product terms $n_r \cdot n_s$, $n_r \cdot n_a$ and $n_e \cdot n_s$. In the common case of hybrid simulation as illustrated in the left part of Figure 3, the models are simulated in one simulation instance without consideration of their distinct temporal characteristics. Two assumptions are given for its validity. Firstly, there are only tight couplings between events and state variables. It is assumed that every event necessarily causes the recalculations of all state variables. Secondly, the output interval is defined as small as possible so that the variable with highest temporal variability can be sufficiently represented. These two assumptions assure the correctness of simulation results and the successful detection of events. Consequently, the above mentioned product terms remain unmodifiable. No improvement of simulation speed can be achieved based on this approach.

The product terms basically describe the couplings between events and variables as well as a uniquely acceptable output interval of all system models. Nevertheless, a hybrid control system is essentially a stiff system in which discrete part and continuous part have strongly different temporal characteristics. A single output interval for all sub models is generally not efficient. Besides of this, not all the events in the discrete sub-system necessarily cause switches between equation systems or stop/restart of continuous integration. Therefore, if a communication mechanism between discrete part and continuous part is given, the integrated simulation can be separated into two parts with a certain synchronization procedure. By doing so, the output interval, the events and the variables have local effects in separated simulations, and if necessary, global effects are guaranteed by synchronization. The separated simulation of a hybrid control system is illustrated in the right part of Figure 3.

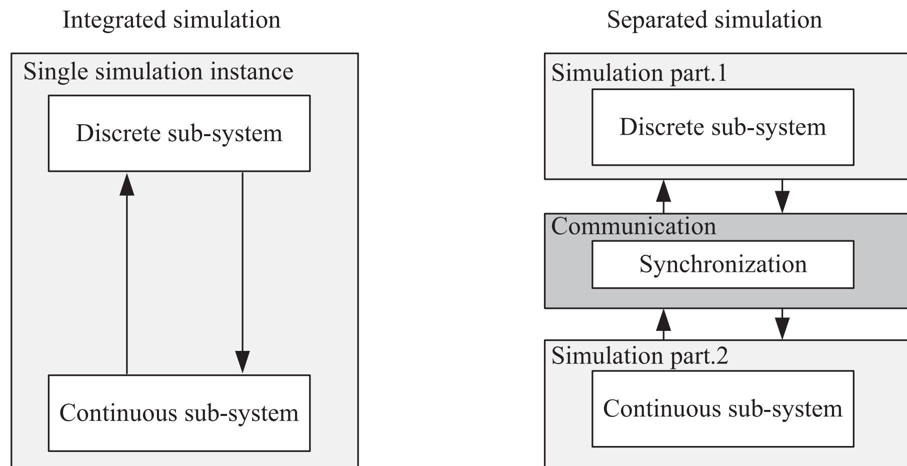


Figure 3. Integrated simulation vs. separated simulation

The basic idea of separated simulation is to improve the simulation performance by cutting off the unnecessary coupling between events and variables. Consider the product term $n_e \cdot n_s$ as an example. It is included both in (3) and (4) which indicate the computational cost. Suppose the number of events in the discrete sub-system is n_{ed} while in the continuous sub-system is n_{ec} and $n_e = n_{ed} + n_{ec}$. The same for the number of state variables with $n_s = n_{sd} + n_{sc}$. Assume 20% of events from the discrete sub-system actually affect the continuous sub-system and the rest only have local effects in discrete sub-system itself. For the integrated simulation, the product term is given by

$$(n_{ed} + n_{ec}) \cdot (n_{sd} + n_{sc}) \quad (6)$$

While for the separated simulation it is

$$(n_{ed} \cdot n_{sd}) + (n_{ec} + 0.2 n_{ed}) \cdot n_{sc} \quad (7)$$

It is clear that (6) takes a greater value than (7). Since different output intervals are selected independently by discrete and continuous sub-systems, similar results can be estimated for the product terms $n_r \cdot n_s$ and $n_r \cdot n_a$.

The separated simulation reveals potential improvement concerning CPU time and memory consumption against the integrated simulation. The fewer events from the discrete sub-system affect the continuous part the more

improvement on simulation speed is achieved. At the meanwhile, the complexity of models is kept so that the accuracy of simulation remains unchanged.

The above discussion is based on the case that the simulation runs on a single processor. The simulation software Dymola discussed in this paper does not support multithreading, i.e. the computational power of up-to-date dual core processors cannot be fully utilized. In the separated simulation, the two simulation instances can be executed on two processor cores. Therefore, the simulation is extended to a quasi parallel simulation further increasing simulation speed.

4 Principles of separated simulation

A successful employment of separated simulation strongly depends on three factors: the allocation of models for diverse simulation instances, the classification of events and the synchronization mode. They are going to be discussed in this chapter.

One important premise for applying the separated simulation is that the system model must be dividable so that one simulation instance is only responsible for a part of the system (Figure 3). To achieve this goal, object-oriented modeling paradigm has to be employed. A system model with object-oriented structure shows not only the hierarchy of models but also the intercommunication between them. It allows the boundary of sub-system easily definable and, because of the existence of the communication mode, the interface for synchronization can be set up with less effort.

The separation of system model is performed in the modeling phase. It concerns the analysis of the system structure and the classification of events. Following notations are used:

- **Event domain:** An event domain is a segment of the system model in which events with similar temporal characteristics are included. An event domain consists of one or more models and is later simulated in one simulation instance. Events in one event domain mainly affect the models in the same event domain. For events requiring communication with another event domain, an interface is needed.
- **Interface:** An interface provides necessary means to set up the communication between different simulation instances, namely, different event domains. By request, events and variables form one simulation can be passed to another through an interface. Each event domain requires one interface.
- **Local event:** local events are the events in one event domain that do not require communication with another event domain.
- **Global event:** global events are the events in one event domain that require communication with another event domain.

Special attention should be put on the classification of local events and global events. To see how the separation of a system model takes place, consider the digital control system in Figure 1 as an example. Firstly, the system is modeled using an object-oriented language, e.g. Modelica. The resulting system model consists of a set of component models which basically describes the structure of the system and the communication mode. The component models may contain various lower level sub-models which describes the internal behaviors of them. Here the detailed timing characteristics and possible events of the component models are investigated. Assume the simulation result is used for stability analysis of an inverted pendulum, typical events of component models and their characteristic values are given in Table 2.

	Events	Timing	Num. of Events
Controller	Execution of control algorithm: begin/end	5 ms	2
	Processing network packet: run/ready	/	5
Communication	Detection medium usage: busy/ready	/	9
	Transmission: begin/end	/	9
A/D converter	Cyclic sampling	1 ms	5
D/A converter	Set new actuation value	1 ms	5
Plant	Only external from actuator	/	1

Table 2. Events of component models

Analyzed is one working cycle, it begins with the request of sensor values and ends by setting actuator values. In the simulation, three types of events are cyclical (light shadowed). The rest are consequent events triggered by the main events from the controller. At the beginning of a working cycle, the controller sends request messages

on sensors. Additional events are triggered in the transmission of network messages. The number of them depends on the topology of communication network and communication medium. By receiving of the request, sensors send back the last sampled values through the network. After processing sensor values the controller sends actuator values to actuators through network and D/A converter. The number of events in one working cycle is given in Table 2. They are estimated from a system model modeled by NC-Library in Modelica. The numbers are not given exactly but represent a valid estimation.

There are more than 30 events in one working cycle but only one actually affects the continuous plant model. The only global event is the setting of the actuation value, the rest are all local events. Two event domains can be defined respectively on discrete sub-system and continuous one. The D/A and A/D converter build up the communication between them. The allocation of them is difficult. If they are classified in the discrete event domain, 10 cyclical events have to be activated on both the discrete and the continuous sub-systems in one working cycle. Due to the frequent synchronization, no remarkable improvement is reachable. If they are included in the continuous event domain, the 10 cyclical events are then local events and have no effects on the discrete sub-system. Certain improvement of simulation speed is achieved. Here a more effective solution is considerable. Until now, explicitly defined sampling events are used for updating/fetching the variables of continuous model. At the meanwhile, the numerical solver also needs to calculate the variables according to a certain output interval. If the output interval is set to the same as the smaller cycle time of A/D and D/A converters, the explicit sampling events can be saved and more improvement is promised. Besides of the existing system models, an interface is required by each simulation instance. The Interfaces are responsible for the synchronization of variables and events through a shared memory. Finally, the separation of system model is illustrated in Figure 4.

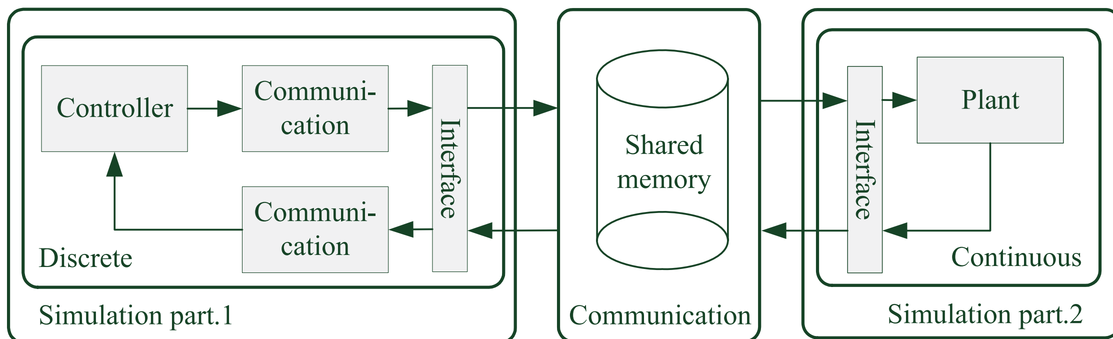


Figure 4. Separation of system model in diverse simulations

The last open question is the synchronization mode. Not only the events and variables need to be exchanged but also the time bases of diverse simulations need to be synchronized. The hierarchy of the hybrid control system shows that the discrete sub-system governs the behavior of the continuous one. Based on this premise, a master-slave synchronization scheme can be applied. In this mode, the master simulation instance starts first. By the occurrence of one global event, it registers the simulation time, pauses itself and commands the slave simulation instance to step to the registered simulation time. After the successful execution of the slave, the master starts again and processes to the next event occurrence. The process is briefly illustrated in Figure 5.

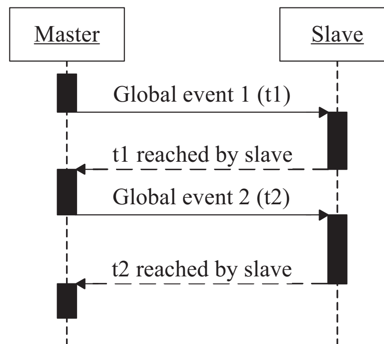


Figure 5. Synchronization mode of separated simulation

5 Implementation of separated simulation in Modelica

Thanks to the well established fundamentals, the implementation of separated simulation in Modelica is quite straight forward. On the one hand, the object-oriented modeling language simplifies the separation of the system model. On the other hand, the external function interface allows the realization of shared memory using high level programming languages. The main tasks of the implementation include the distinct interface designs for master/slave and the realization of shared memory for exchanging variables.

The interfaces are integrated in diverse simulation instances. Because there is no additional arbitrator for the synchronization, the interfaces should take over this task. This fact also leads to the distinct designs of master and slave interfaces.

The master interface (Figure 6) connects the discrete sub-system. Three connectors are used to establish the connections to existing models. The *"controller to process"* connector exports the variables which need to be sent to the continuous simulation instance, e.g. actuator setting value. The *"process to controller"* connector imports the variables from continuous simulation. The export and import are running on the basis of shared memory. The third connector *"trigger events from controller"* gathers all the global events selected from the discrete sub-system. The simulation procedure with master interface is illustrated in Figure 7. A cyclical scan is performed on the connected events. If any event is activated, the master interface pauses the simulation of the discrete sub-system by a while loop. The relevant variables as well as the current simulation time are passed to the slave simulation instance. In the while loop, it keeps polling the status of the slave simulation. If the slave simulation is successfully finished, the simulation of the master is resumed after importing variables. The procedure is repeated until the end of simulation is reached.

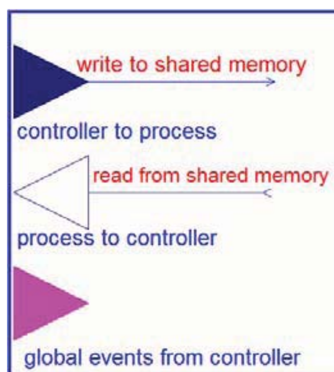


Figure 6. Modelica master interface

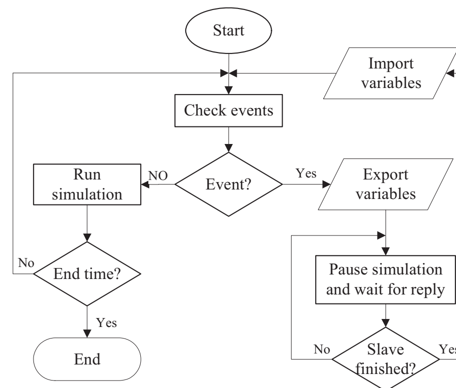


Figure 7. Simulation procedure with master interface

The slave interface (Figure 8) connects the continuous sub-system. Two connectors are given for importing and exporting variables. They are connected with the corresponding connectors of the master interface through shared memory. As shown in Figure 9, the slave simulation instance waits until an activation command is given by the master. After importing necessary variables from the master simulation, it decides either to simulate to the given simulation time or to the end time of the simulation. In the first case, simulation is advanced to the given simulation time, namely the occurrence of event in master simulation. After reaching this time point, variables are exported to shared memory and the slave simulation is paused until the next activation. As a matter of fact, the end times for both simulation instances should be identical.

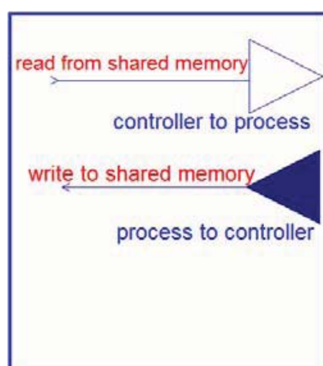


Figure 8. Modelica slave interface

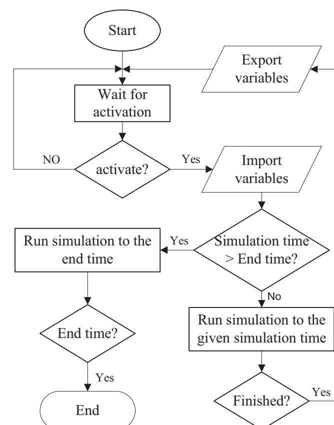


Figure 9. Simulation procedure with slave interface

As mentioned before, two simulation instances exchange variables through shared memory. The basic idea is as the same as a data base. Because the simulations store the complete variables during whole simulation separately, only the values of variables at the occurrence of events need to be exchanged. The shared memory is implemented using Dynamic-Link Library (DLL). The DLL approach enables different simulation instances using the same memory section to exchange the variables. The variables in the shared memory are identified by index numbers but not names. Two functions are provided for the manipulation of the shared memory. The function *SetSharedMem(index, value)* stores a value under the index number, while the function *ReadSharedMem(index)* fetches the variable according to the index number.

6 Case study

The subject of this case study is to demonstrate the benefit of the separated simulation against the conventional integrated simulation concerning memory consumption and CPU time. The tested model is given in Figure 10. It describes a trajectory control of a robotic arm using NCS structure. The embedded controllers are connected using 10 Mbps fully switched Ethernet. The robotic arm consists of five revolute joints connected to five drive axes. Each drive axis consists of an axis controller, three sensors providing information about angel, angular velocity and torque, a DC motor and gearboxes (see the bottom-right in Figure 10). The sensor values and setting voltage are periodically (0.1 ms) sampled/updated by the A/D and D/A converters of the axis controller. The trajectory plan controller periodically (10 ms) sends request messages to all the axis controllers through the network. The axis controllers reply with the newest sampled sensor values. When all sensor values have been received by the trajectory controller, it begins to compute the new torque references according to a defined control algorithm and sends them to axis controllers which compute the respective setting voltage accordingly. The robotic arm is modeled using the Modelica Multibody library while the controllers and communication components are modeled using the Modelica NC-Library.

The separated simulation of the robotic arm is illustrated in Figure 11. The axis model contains both discrete and continuous sub-models thus a detailed division is required. The sensors, motor and gearboxes are extracted to build a new model named *slaveaxis*. The axis controller is integrated in the new model named *masteraxis*. According to the principles from chapter 4, the periodical sampling devices are eliminated from it. The necessary global events include the request of sensor values and the updating of actuator values. They are extracted from the *masteraxis* model and connected to the events input port of the master interface. Similarly, necessary variables are selected and connected to the interface models.

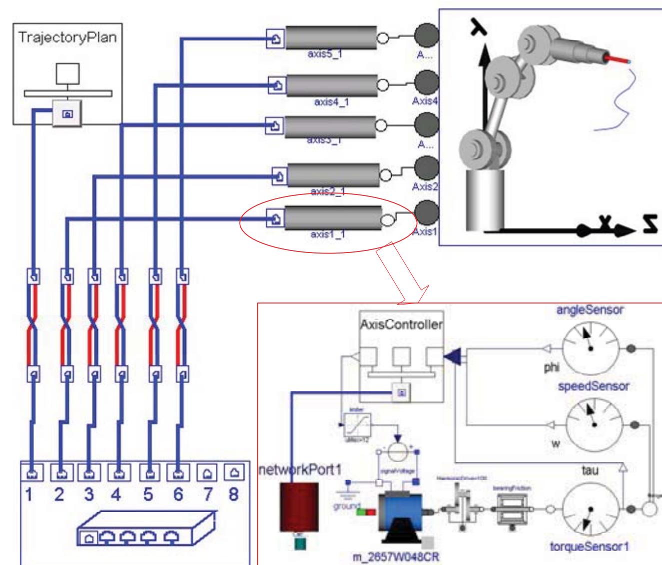
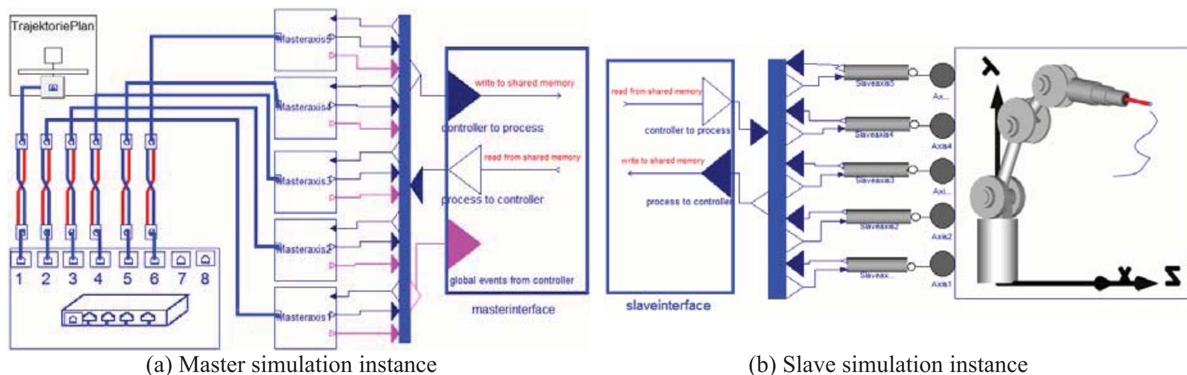


Figure 10. Integrated simulation of robotic arm



(a) Master simulation instance (b) Slave simulation instance

Figure 11. Separated simulation of robotic arm

Simulations are performed for 1 second simulation time under the same hardware and software conditions as described in Chapter 2. The used integration algorithm is Lsodar with 10^{-6} tolerance. The main model properties, setting of solver and computational cost are listed in Table 3. We notice that the sum of auxiliary variables in the separated simulation is slightly larger than in the integrated simulation, this is due to the additional connectors and variables in the interface models. The setting of output interval also differs from each other. In the

integrated simulation, the sampling events already provide sufficient resolution thus no explicit output interval is required. In the master simulation instance, because the variables are registered on the occurrences of events, the output interval is again irrelevant. On the other hand, due to absence of sampling devices in separated simulation, the output interval is set to the same as the sampling rate in slave instance. The comparison shows that the separated simulation is about 2.5 times as fast as the integrated simulation on single core processor. If a dual core processor is utilized, up to 6 times acceleration is achieved. Last but not least, less memory consumption is achieved by the separated simulation. Both approaches get the same simulation results under the same control algorithm. Consider about the consumption of the computational power by the polling operations of CPU and context switching in the synchronization, the result conforms to the analysis in chapter 3.

	Integrated simulation	Separated simulation	
		Master	Slave
Num. state variables	93	78	15
Num. auxiliary variables	4400	1226	3243
Num. time events	10000	100	0
Num. state events	3288	3420	626
Num. result points	26576	7042	10626
Output interval (s)	1	1	1e-4
Size. result file (MB)	482	27	160
CPU time (s)	135	54 (single core) 20.6 (dual core)	

Table 3. Comparison of two simulation approaches

7 Summary and outlook

In this paper, a separated simulation approach for hybrid control systems is presented. Basic idea is to allocate the system models to two separated simulation instances according to the native hierarchy of hybrid control systems. The potential improvement of separated simulation is qualitatively demonstrated based on the analysis of working principles of numerical solvers. An implementation of separated simulation has been done on the platform Modelica/Dymola in this paper. It allows the conversion of an integrated simulation problem to a separated simulation with minimum modification. The benefit of this approach has been proved on the example of trajectory control on a robotic arm. The result reveals that the separated simulation outperforms the integrated simulation in CPU time and memory consumption. Furthermore, the separated simulation fully utilizes the computational power of the dual core processor technology. A quasi parallel simulation is realized by this approach.

Future work includes a graphic user interface design for simplifying the execution of separated simulation. Furthermore, it has been observed that the computational cost of the polling operation in the synchronization takes a large part of the overall usable CPU time. I.e. the polling operation of one simulation instance hinders the execution of the other one. Methods are being investigated to reduce this overhead.

The Modelica libraries presented in this paper are available at www.eit.uni-kl.de/frey.

8 References

[1] F. Wagner, L. Liu, G. Frey, *Simulation of Distributed Automation Systems in Modelica*, In: Proc. 6th International Modelica Conference, Bielefeld, Germany, pp.113-122, Mar. 3-4, 2008.

[2] F. Cellier. *Combined discrete /continuous system simulation by use of digital computers: techniques and tools*. PhD thesis, ETH Zurich, Zurich, Switzerland, 1979

[3] *Dymola –User's manual*. Dynasim AB. Research Park Ideon, Lund, Sweden, 2002.

[4] R. M. Fujimoto, *Parallel and distributed simulation systems*. Wiley Interscience, 2000

[5] T. L. Vincent, W. J. Grantham, *Nonlinear and Optimal Control Systems.* Wiley Interscience, Juni, 1996

[6] K.E. Brenan, S.L. Campbell and L.R. Petzold: *Numerical Solution of Initial Value Problems in Differential--Algebraic Equations*. Elsevier Science Publishers, 1989

[7] A.C. Hindmarsh: *ODEPACK, a systematized collection of ODE solvers*. Scientific Computing, edited by R.S. Stepleman et al., North-Holland, Amsterdam, 1983.