Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

C. Schlegel, R.Finsterwalder, H. Olsson
Schlegel Simulation GmbH; BU München,Germany; Dynasim AB, Sweden
Using Dymola generated C-Code in specialized Client / Server Simulation Environments
Not published

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Using Dymola generated C-Code in specialized Client / Server Simulation Environments

Clemens Schlegel
Schlegel Simulation GmbH
Meichelbeckstr. 8b
D-85356 Freising

Reinhard Finsterwalder
University of the Federal
Armed Forces Munich
D-85577 Neubiberg

Hans Olsson
Dynasim AB
Research Park Ideon
S-22370 Lund

Abstract

The modeling and simulation tool Dymola can generate C-code from a Modelica simulation model. We investigate the usage of that code in different client / server environments with the simulation running on a server and a simulation GUI running on a client. The Dymola generated model code (server) has been run under Windows 2000, Linux and the realtime operating system QNX. Clients have been implemented in Matlab/Windows 2000, C++/MFC under Windows 2000 and Windows Mobile, and Java/ AWT. We used DDE and TCP/IP as communication protocols. As a demo simulation a model of a cold rolling mill with 462 unknowns and 33 states has been used. A multithreaded, shared memory buffer mechanism for the communication turns out to be a good approach for distributed simulation setups including realtime simulators.

1 Introduction

The modeling and simulation tool Dymola [1] can generate C-code from a Modelica simulation model. That C-code may be used in the tool's own simulation environment, in which case the generated code plugs in seamlessly. It may also be integrated in a different, user provided environment. In this paper we will have a look at the second case.

A simulation environment to run model specific code has to provide mainly four functionalities:

1. Numerical integration, if the integrator is not part of the generated code and the model contains differential equations.
2. Simulation control. The minimum functionality is to start and stop the simulation run, it may comprise synchronization
3. Parameter handling: load, inspect, change and save parameters.
4. Trajectory handling: display, evaluate and save simulation results.

Often the model code is run on the same hardware and software platform as the simulation environment. If not (the case we are discussing here) the communication between them is a main issue.

Porting the model code from the model development system to another platform is mostly driven by special needs like cosimulation of different simulation systems, hiding a model completely from the user, providing a specialized user interface, embedding the model in a special environment like a dedicated training simulator, or an animation system, or running the model on a specific realtime platform.

In this paper we will have a look at DDE and a more closer look at TCP/IP for communication. We run the model and the numerical integration on different servers, and simulation control, parameter handling and trajectory display on several clients. The different client/server configurations used are given in table 1, the corresponding hardware in table 2.

For the configuration Win/DDE we applied Dymola's optional feature to link the model code as a DDE server [2] and used Matlab (could be any other appropriate Windows application as well) as DDE client [3]. For the other server configurations we used the Dymola standalone option to set up an autonomous model code (including an integration algorithm) and instrumented that code with TCP/IP

communication. Finally, the QNX server [4, 5] is a true realtime system (e.g. interrupt latency < 4 µs [6]).

2 The simulation test model used

All the simulation tests have been performed using a simulation model of a cold-rolling mill. The model has been developed for investigation of the rolling process in order to design an innovative automation system for such plants [7]. The project focused on thickness control and production throughput. Therefore the modeling scope does not consider flatness, crown and width control.

Figure 1 gives a schematic overview of a typical single stand cold rolling mill. The metal strip (2) is unrolled from an uncoiler (1), which feeds the strip into the roll gap (4) via a deflector roll (3). The force on the deflector roll is measured to control the strip tension via the uncoiler drive. Several back-up rolls (6) support a set of two work rolls (7), which directly act on the strip to reduce its thickness, which is measured after the roll gap. The required force to control the roll gap amplitude is generally generated by hydraulic cylinders (not shown in figure 1). The strip is then wound up on another coil (5). Together with the coiler drive an other deflection roll is used for tension control on the output side.

Modeling and simulation of such plants imposes several problems:

- Uncoiler and coiler radii are changing discontinuously during the rolling process.
- Friction effects of the strip and deflection rolls may cause tension oscillations.
- Friction in the hydraulic cylinder and in the rolling mill stand has a strong influence on thickness control and therefore needs to be modeled with high fidelity.
- Elastic and plastic deformation effects have to be considered in the roll gap. The resulting equations lead to algebraic loops in a nonlinear implicit formulation.
- Due to the mentioned nonlinear implicit algebraic relations model initialization is not trivial.
- The complete thickness reduction of a single coil is performed in several passes and takes up to 15 minutes. Since the required integration step size is quite small (typically 0.1 to 0.5 ms), simulation speed is important.
- For operator training a realtime simulation of the overall plant is desirable.

The demo model has 54 components, 791 variables (33 of them are selected as states by the Dymola translator), 462 unknowns and 329 non-trivial equations. Using a PC with medium performance (table 2) 0.55 ms computing time per 1 ms integration step is required under the realtime system QNX.

3 Communication Protocols

Because of the built in functionality of Dymola and Matlab we first tried DDE for client / server communication. That combination offers a comparatively easy solution. Later in the project we switched to TCP/IP. To make sure that the client does not miss any trajectory data of the simulation server we used a multithreaded buffering scheme described below.

3.1 DDE

The salient point of using Win/DDE is the performance of the DDE communication between the model code and the user interface. DDE is not known for superior performance and requires special provisions if partial data loss is not acceptable. If e.g. the GUI is busy due to some user interaction, data sent by the DDE server gets lost, there is no built-in buffer or client/server handshake mechanism. Using the handshaking defined for DDE_ADVISE [8] is not always possible, because it is not accessible from e.g. the Matlab-client and it degrades performance due to lack of a built-in buffer.

NetDDE extends DDE for client/server applications across a network. It can be used in a LAN / WAN because it’s based on NetBios what might be run over TCP/IP. However, NetDDE is mainly available for Microsoft operating systems. There are other (proprietary) implementations available but these are not part of the respective operating systems.

To overcome the data loss problem the mentioned multithreaded buffering scheme could be applied as well, but that requires to re-program the complete communication (both client and server) from scratch. Instead of doing so we switched to TCP/IP.

3.2 TCP/IP

In order to cover a great variety of different platforms and for performance reasons we mainly used the TCP/IP network protocol. It is integral part of all current operating systems including realtime systems. TCP/IP guarantees the correct transmission of data packages from sender to receiver. Data packages are received in the order they are sent. TCP/IP supports the communication of distributed processes running in a heterogeneous computer network. E.g. the simulation can be run on a QNX computer in realtime, while the client runs on a Microsoft PC.

4 Client / Server Implementation

4.1 Server implementation

The simulation computer (server) runs two unsynchronized processes, the simulation process and the data communication process. The simulation process runs at a fixed sampling rate, e.g. 1ms. The output of each integration step is written to a FIFO buffer (first-in-first-out) which is implemented as shared memory, what is the most efficient way of interprocess communication [10]. The communication process asynchronously reads that buffer and sends the data to the client computer for display and further processing. In order to avoid dead-locks a semaphore mechanism is established to control the concurrent access to the shared memory buffer [10]. The size of the buffer is chosen large enough to hold simulation data for a longer, user-defined time intervall t_{buffer} (e.g. 0.5 s). By that, delays on the client-side due to interrupts of the operating system or user interaction can be handled without any loss of data. The buffer size is adopted to the specific data rate imposed by the simulation model dynamics and the communication bandwidth. It can be computed by:

$$buf_{size} = \frac{t_{buffer}}{t_{step}} \cdot n_{signals} \cdot sizeof(datatype)$$

where

- t_{step} is the step-size of the integration,
- t_{buffer} is the desired time intervall
- $n_{signals}$ is the number of signals to be transmitted
- *sizeof(datatype)* is e.g. 4 Byte for float on 32-Bit CPUs.

For the demonstration model with $n_{signals}$ = 9, t_{step} = 1 ms and t_{buffer} = 0.5 s, a buffer size of ~ 18 kByte is required.

The data communication process on the server is client-driven. The server is listening on a dedicated network port. When a client request arrives, a new thread is created that handles the data communication with this client while the main process is waiting for further client requests. This multi-threaded implementation allows the access of multiple clients to the simulation process using a master client for simulation control (figure 2). For multi-client access the server load increases only slightly, because only the communication buffer is duplicated and not the simulation code.

4.2 Client Implementations

Data exchange between the server and the client is done on explicit client request only. For visualization and user interaction purposes it is sufficient to update the display at a rate of 25 frames/sec, what means the client requests data from the server every 40 ms. When the server receives a client request, the server sends the whole actual content of the shared memory buffer wrapped in TCP/IP packages. Thus multiple time instants are transmitted in one package what's much more efficient than sending single time instants in small packages.

In order to demonstrate the described mechanisms we have implemented several clients using different programming languages (see table 1).

4.3 Matlab Client

Figure 3 shows the Matlab client in action. The Matlab Mex-API is used to interface to our communication procedures. Three functions are provided wrapping the whole buffered communication:

- *mxTCPIPOpen* – connects to simulation server
- *mxTCPIPClose* – terminates communication
- *mxTCPIPCom* – sends/receives data

If the GUI is busy due to some user interaction, the online plot is stopped until the user releases the mouse. Since the simulation output is buffered on the server, no information is lost, given the buffer is large enough. A better result can be achieved by splitting GUI and online visualization in two processes. However, Matlab does not support multithreading. A practical workaround is to run two Matlab systems at the same time. In the first instance the control panel is running. The second instance is

started and controlled by the control panel via the Matlab engine.

4.3 Java Client

Figure 4 shows the Java client in action. Although the execution of Java bytecode is done by an interpreter, the performance is very good. Java programs are compiled into the machine-independent bytecode, therefore Java programs can be run on all operating systems without changes in the source code or recompile [11].

Java programs can be incorporated as applet into HTML pages. When a web-browser finds an applet tag in the current html page, it automatically downloads the corresponding bytecode from the web-server to the user's host computer and executes the applet there. This feature provides a simple mechanism to deploy software over the Internet without having to be concerned about installing, configuring or maintaining software on the client computer.

4.4 PDA Client

Figure 5 shows the PDA client in action. The application is developed with eMbedded Visual C++ [9]. In our setup the PDA accesses the simulation server through a wireless LAN. Due to the limited computing and graphics power, data buffering on the server-side is essential. Nevertheless, the quality of the visualization is sufficient for monitoring and supervision purposes. It is planned to extend the software to communicate via GPRS and UMTS, too. In addition we will look at using the Java client on a PDA.

A further PDA application beyond our rolling mill process control example could be a test driver doing ECU calibration work (e.g. tuning the parameters of a electronic gearbox control unit). To pre-check a new set of parameters remote access to a detailed vehicle simulation is quite useful.

4.5 Performance issues

Due to multithreaded buffering the communication parts of the overall computing load are quite low. The main limiting factor for the communication bandwidth is the network capacity.

For a Matlab GUI client DDE offers a quick solution. It is not high performance, but quite sufficient if partial data loss is acceptable and the setup requires only moderate data rates. We achieved roughly 1kHz data rate for that setup.

We have tested our TCP/IP approach in a fast Ethernet network (100 MBit/s) and in a IEEE 802.11b WLan (11 Mbit/s). Tests showed that no data gets lost. The results are independent of the operating system of both the client and the server. Using average performance computer equipment (see table 2) we achieved a data rate of 50 kHz without any tuning running both client and server under Windows. The use of network equipment with higher capacity (GigaBit Ethernet, 54 Mbit/s WLan) offers enough margin for quite high data communication rates.

Obviously a performance limitation is given by the client graphics speed. Since the communication is client driven, it may occur that the frame rate decreases below 20 frames/s. Nevertheless if the server-side buffer is large enough it can be guaranteed that all data arrive at the client. For realtime and hardware-in-the-loop simulations, the separation of simulation and data communication into independent processes is essential in order to avoid that a client slows down a fast server.

5 Conclusion

Based on the described communication scheme a prototype of a framework for distributed client/server simulation experiments has been build. Although the current implementation is still a-proof-of-concept which lacks some of the features which are required for professional use, the proposed communication scheme turns out to be a good approach for distributed simulation setups including realtime simulators.

6 References

[1] www.dynasim.se
[2] Dymola 5 User's Manual. Dynasim AB, Lund Sweden. 2004
[3] www.mathworks.com
[4] www.qnx.com
[5] Rob Krten, Getting started with QNX Neutrino. A Guide for Realtime Programmers, Parse Software Devices, 2001.
[6] QNX Neutrino 6.2 RTOS evaluation report. www.dedicated-systems.com, 2002
[7] Andreas Kroll, Andreas Vollmer, Industrial^{IT} für Kaltwalzwerke. Die nächste Generation der Kaltwalzwerksautomation. ABB Technik 4/2004.

[8] msdn.microsoft.com
[9] Steven Makofsky, Pocket PC Network Programming. Addison Wesley, 2003.
[10] Bill O. Gallmeister, POSIX.4. Programming for the Real World. O'Reilly, 1995.
[11] J. Gosling, B. Joy and G. Steele, The Java Language Specification. Sun Microsystems Inc., Addison Wesley, 1996.

7 Acknowledgement

The authors would like to thank Mr. Benno Rieger, studying computer science at the university of applied sciences Munich, who did most of the coding and testing during his industrial training.

Table 1: Client / Server configurations investigated (if not stated explicitly TCP/IP is used for communication. For DDE we run client and server on the same computer, for TCP/IP on different computers).

Clients:	Matlab / DDE	Matlab	Win MFC32	Win Mobile / WLan	Java
Servers:					
Win / DDE	X				
Windows		X	X	X	X
Linux		X	X	X	X
QNX		X	X	X	X

Table 2: Hardware and OS-software used

	Hardware	Operating System
Server	Intel Pentium M, 1.4 GHz, 512 MB	Microsoft Windows 2000 Professional Debian Linux 2.6.10 QNX Neutrino 6.2
PC Client	Intel Pentium M, 1.4 GHz, 512 MB	Microsoft Windows XP Professional
PDA Client	Dell Axim X5, Intel Xscale 400 MHz, 64 MB	Microsoft Windows Mobile 2003

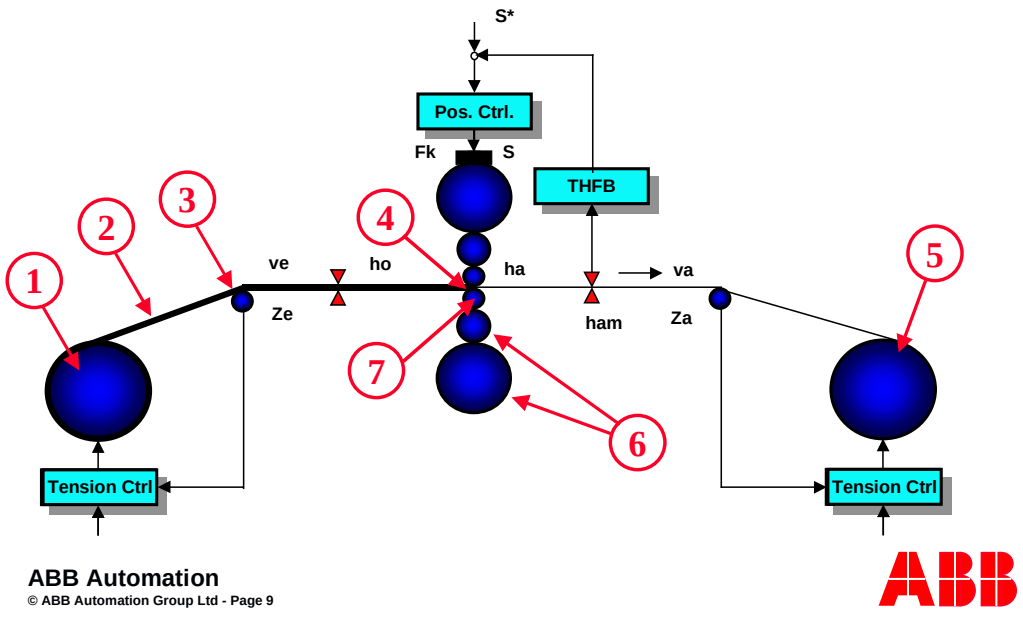

Figure 1: Cold rolling mill schematic overview

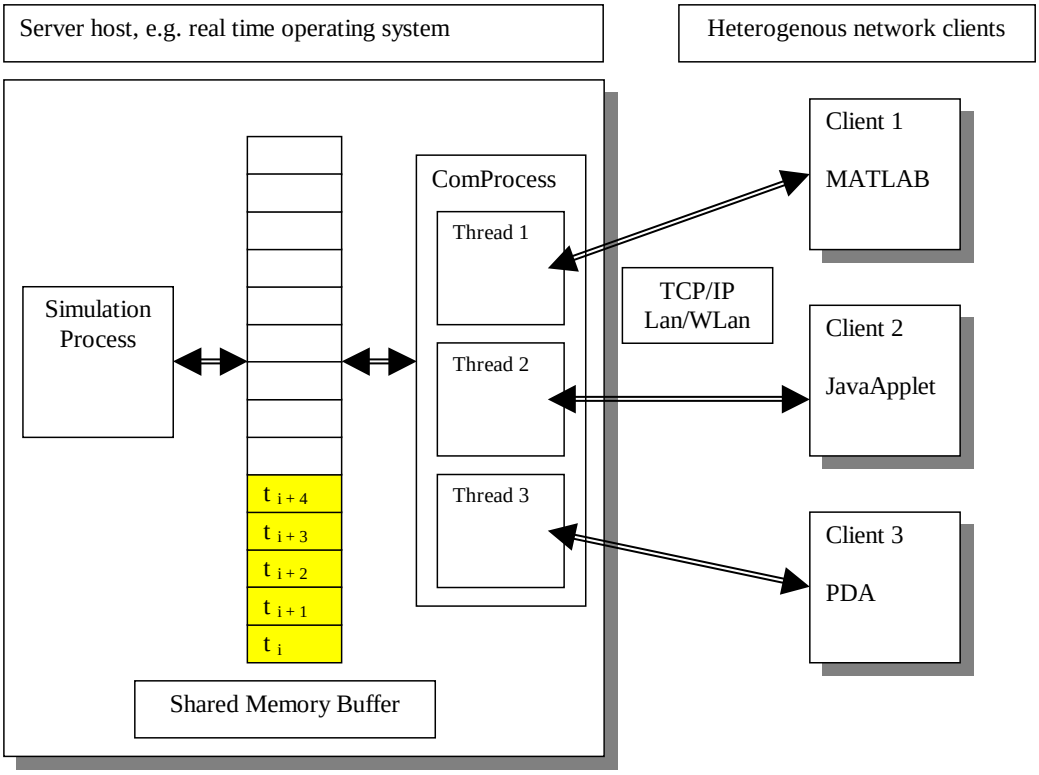

Figure 2: Client / Server implementation

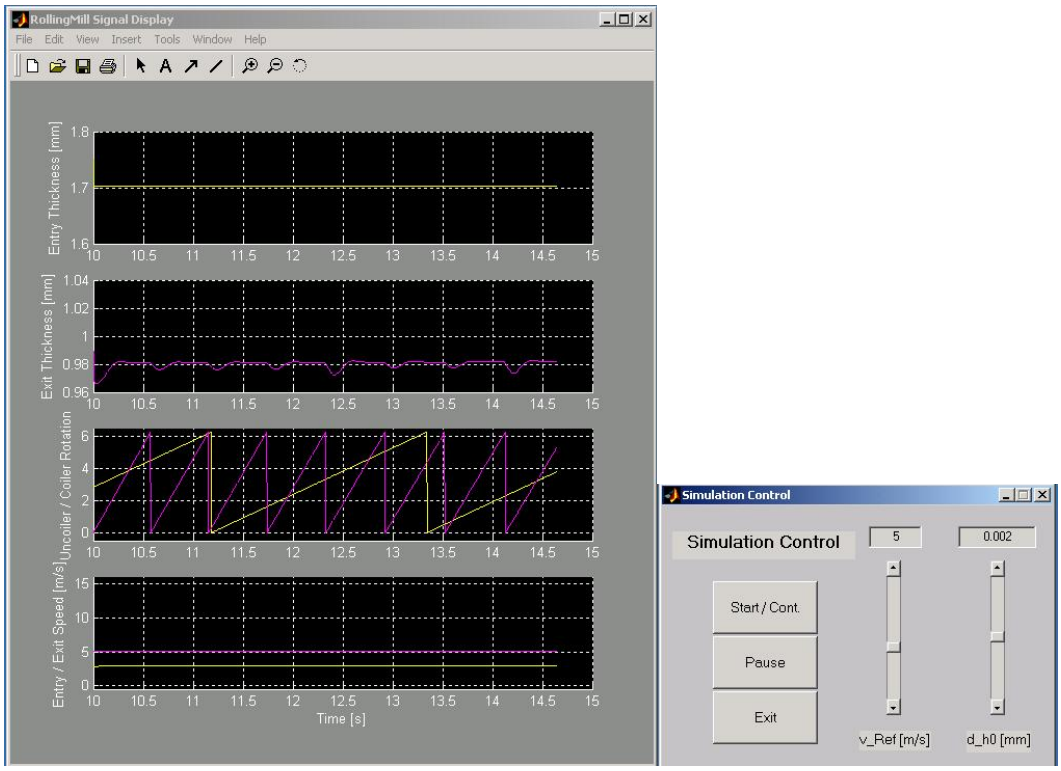

Figure 3: Matlab client (same GUI for both DDE and TCP/IP)

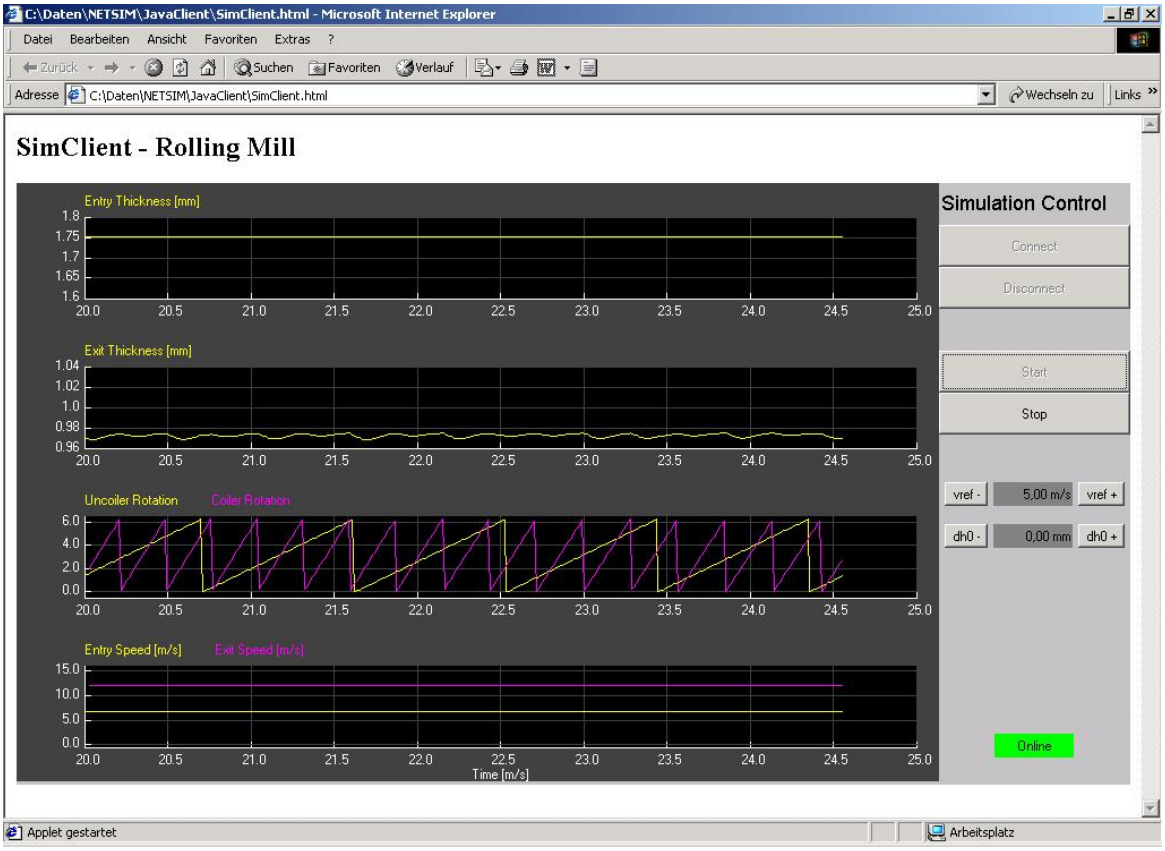

Figure 4: Java client

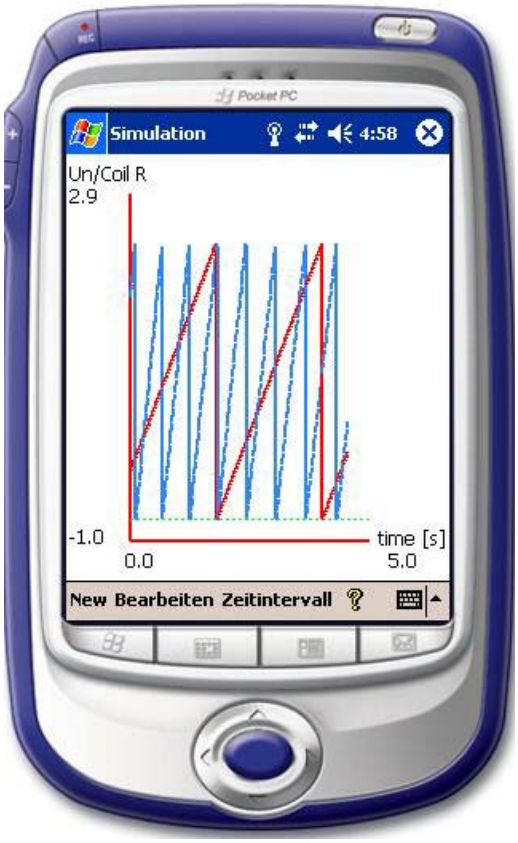

Figure 5: PDA client