ABCM Symposium Series in Mechatronics - Vol. 2 - pp.589-596

Ibex - A Framework for Hardware in the Loop Simulation

Patrick Büchler
Institute of Electronics, University of Applied Sciences of Central Switzerland (FHZ)
pbuechler@hta.fhz.ch

Alan Ettlin
Institute of Electronics, University of Applied Sciences of Central Switzerland (FHZ)
pbuechler@hta.fhz.ch

Bradley J. Nelson
Institute of Robotics and Intelligent Systems, Swiss Federal Institute of Technology Zurich (ETH)
bnelson@ehtz.ch

Abstract. *The design of control-algorithms for mechatronic systems is in general difficult and error-prone. Therefore, a common practice is to use Hardware in the Loop Simulation (HILS) for an incremental test and development process.*
Ibex is a real-time simulation software framework for HILS which has been designed specially for rapid prototyping. A key feature of Ibex is its modular architecture. One type of module can simulate physical systems. For example we have integrated a powerful rigid body dynamics engine into Ibex. Other modules, the so called observers can analyze the state of the simulation at runtime. Implementations thereof range from a simple logfile to sophisticated 3D visualizations of the simulated system. Tools for the detailed inspection and analysis of simulation parameters are also provided. In order to simulate complete mechatronic systems, Ibex additionally can include virtual sensor, actuator and low-level controller modules.
The power of the framework is demonstrated by describing two industry projects in which we have used Ibex for HILS. The example of a parallel-kinematic delta robot illustrates how a standard PC can simulate a non-trivial geometry and still keep up a high bandwidth communication with the real controller of the robot. The second showcase is the simulation of a complete elevator system which visualizes the interaction of a number of complex mechanical parts within the simulation

Keywords: *hardware in the loop simulation, HILS, real-time simulation, robotics, software framework*

1. Introduction

In general, the development of mechatronic systems can be divided into three different categories, the development of the mechanical properties of a system, the design of appropriate control algorithms and the programming of higher order application logic. Members of a development team therefore usually come from the broad disciplines of mechanical engineering, electronics and computer science. Successful projects require a parallel and incremental test and development process, that involves all team members from the very beginning on. Designing control algorithms, for example for industrial or mobile robots, is complicated and error-prone. To alleviate the task, a common practice is the use of simulators to test and develop the control algorithms and the application logic. so called Hardware in the Loop Simulations (HILS) connect control hardware to a simulator which provides it with the necessary input data about the state of the environment and the entity being controlled. The controller receives data and issues commands using its normal input and output channels. Therefore, it is completely transparent for the controller that the communication does not take place with real sensors and actuators but with simulated replacements. For certain control and robot models simple state-machines are sufficient as simulators. However, the need for a *complete* simulation arises as soon as continuous data about the condition of the system and its environment is of interest. The Ibex framework provides a real-time environment for HILS projects which require a continuous simulation. The following section describes the architecture of Ibex in detail and shows how it can be used for HILS. After having presented the architecture, two concrete projects using Ibex are shown as well as future plans and other possible applications for the Ibex framework are outlined.

2. Architecture

The aim has been a system which fulfills all the requirements listed in tab 1. The framework architecture is strictly object oriented and makes extensive use of design patterns, as described for example by Gamma et al.(1992). This leads to an easy comprehensible architecture and reduces the possibility of errors. Additionally, it improves the quality of code in general. Since it was not the goal to reinvent the wheel, external libraries have been integrated wherever possible.

Figure 1 shows a simplified UML diagram of the core classes of Ibex, their relationship and their most important methods. In the Ibex terminology the whole simulation task to be executed is called a system. Therefore, according to the used naming convention, the main class of our framework is called `CIxSystem`. The behaviour of the system is controlled by a System Controller, the corresponding class in the framework is `CIxSystemController`. Ibex is multi-threaded, this

Table 1. The requirements for the Ibex real-time simulation framework

Criteria
The software must be easily extendable.
The framework should allow a fast integration of interfaces to external hardware.
A faithful simulation of the mechanical properties of the environment and the devices to be tested must be possible.
A graphical user interface must be provided.
The state of a simulation has to be representable and analysable with a number of different utilities ranging from tabular data structures to intuitive 3D-Graphics.
The hardware requirements have to be moderate. A standard PC with a 3d-graphics accelerator must suffice. Consequently our test computer was a 1.7 Mhz Centrino Laptop with ATI Radeon 9200 graphics chip.

allows a faster execution time and alleviates the connection of external hardware or other software systems. The thread class is called `AIxThread`. Mutual exclusion between different threads is guaranteed by the class `CIxMutex`. `AIxEvent` and the event handler implement in `CIxEventHandler` allow the framework to react quickly on asynchronous events, as for example a command issued by the controller to be tested. A more detailed description of threads and the event handler is given in section 2.5.

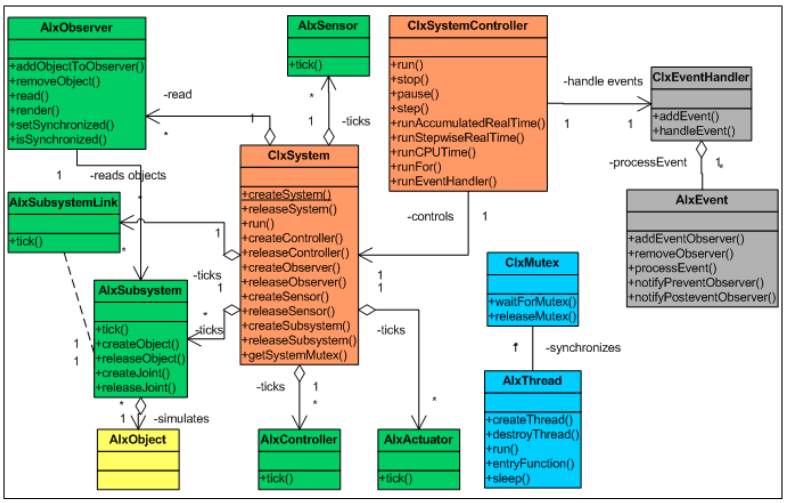

Figure 1. UML Diagram of the most important classes

An analysis of common real-time simulation tasks in the context of Ibex has revealed that they usually consist of five different parts. For each of these parts, an abstract interface with the basic set of methods and variables required by the simulation has been created. Table 2 lists those five components and describes them in short term.

Table 2. The five different components of a common Real-Time Simulation tasks

Component	Interface	Description
Subsystem	*AIxSubsystem*	Subsystems contain the actual solvers used to simulate part or all of the physical phenomena which are of interest within the simulation application.
Observer	*AIxObserver*	Observers visualise the current state of the system. This can be performed by a 3D-graphics engine, a tabular data-representation or a log-file etc..
Controller	*AIxController*	In the terminology of ibex, Controllers are devices used to steer the behaviour of actuators within the simulation. I.e. the controllers for the motors of a robot.
Sensor	*AIxSensor*	Virtual sensors are used to simulate the behaviour of real sensors within the simulation. They provide the input data required by the real control-hardware that should be tested.
Actuator	*AIxActuator*	Actuators represent characteristics of motors which are used in the simulation.

In the next subsections the functionality of these five components is inspected in more details and it is outlined how

the main loop in the class `CIxSystem` brings all the different parts together. At the end of this chapter a subsection about multi-threading will shortly describe how asynchronous communication within the simulation and with external entities like control hardware has been implemented.

2.1 Subsystems and Rigid Body Dynamics

A subsystem is a numerical solver for a certain class of physical problems. Ibex has been designed especially for HILS tasks. Therefore, we assume a subsystem being a rigid body dynamics engine, because these programs are usually real time capable and offer faithful representation of the classical rigid body mechanics as already described by Newton. Typical examples for such simulation engines are Novodex (Novodex 2005) or ODE (ODE 2005). Most of the algorithms used for rigid body dynamics are based on work as the one by Baraff (1989, 1991, 1996) or Mirtich (2000). Subsystems provide the method `tick` to execute a given timestep of a simulation. Objects, represented by `AIxObject`, are the rigid bodies interacting within the simulation. Subsystems offer the possibility to add forces and torques to them. Objects have properties like mass, density or the inertia tensor and they are supposed to have a geometry called the collision shape.

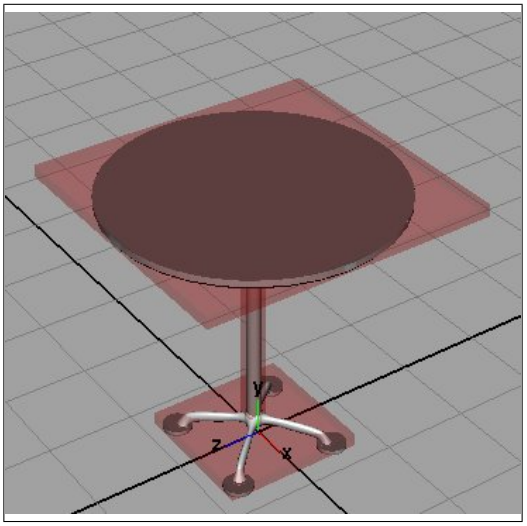

Figure 2. The Collision shape of a table. It is a combination of a rectangular box for the bottom parts, one for the table top and one for the table legs.

The collision shape is the amount of space occupied by the object within the simulation. It can be represented by an arbitrary polygonal 3d-mesh with very high precision. Unfortunately, this is not efficient when it comes to collision detection. Therefore, Ibex offers so called primitives, which are boxes, spheres and capsules. The most efficient way to represent the collision geometry of an object is to combine a number of those primitives. As an example, the collision shape of the table shown in fig. 2 is represented by three boxes. Objects are connected by joints. They simply reduce the degrees of freedom between two given objects. A number of predefined joints simplify the development of complicated mechanical setups.

2.2 Actuators and Sensors

Actuators and sensors are usually the means of interaction of a control unit with its environment. I.e. a robot obtains information about the surrounding space by means of a camera. It then analyses the data and makes a decision about its next motion target. To reach the target, the robot applies a certain torque to its actuators. To be able to model this behaviour in the simulation, actuators as well as sensors have been introduced into Ibex. The model is very simple, every `AIxActutor` or `AIxSensor` has to implement the `tick` method. Within the tick method it is free in its action. According to its requirements it can either apply forces or torques to objects within a given subsystem or read data like positions and orientations. Needless to say that in a real time simulation it is not possible to accurately simulate the behaviour of an actuator. Instead, Ibex encourages the use of a special predefined actuator, which can be provided with a lookup table containing the characteristics of the desired motor. The table simply connects a given current input to an output torque. With this strategy, the implementation of different motor models is possible without writing one line of code.

2.3 Observers

Observers offer the possibility to analyse the behaviour of the simulation. A very nice and intuitive way is offered by modern 3d-graphics engines. Ibex has integrated a fully implemented 3d-engine basing on the open source project

Irrlicht (Gebhardt 2005). Every simulation done with Ibex automatically profits from this feature. Details on the already implemented functionality are given in section 2.7. 3d-graphic observer are usually not synchronized. This means they are not called by the main loop at every simulation step but run in a separate thread and update themselves whenever necessary. This guarantees that the simulation execution is not slowed down by the rendering process. If a simulation needs a large amount of processing power to keep up to the real-time requirements, the 3d-engines can be reduced in priority or turned off totally.

Another type of observer are spreadsheet-like data-structures. Ibex already offers predefined functionality to extract data like position, velocity and acceleration of every object at each timestep (see also section 2.7). This type of observer is often used to analyse the behaviour of the simulation or the tested control units. In addition this observers can be configured so that they log only when a certain event, like a collision between two objects, occurs.

2.4 The system class and the main loop

One step of the simulation is executed by the private method `run()` in `CIxSystem`. This is called the main loop. Its implementation is described in in pseudo-code in the following code snippet:

```
if (not system.isPaused())
{
    for(listOfAllSensors)       //go through all sensors and
    {                           //execute tick
        controller.tick();
    }
    for(listOfAllControllers)   //go through all controllers and
    {
        controller.tick();
    }
     for(listOfAllActuator)      //go through all actuators and
    {                           //execute tick
        controller.tick();
    }
    for(listOfAllSubsystems)    //go through all sensors and
    {                           //execute tick
        controller.tick();
    }
}
for(listOfAllObservers)         //go through all observers and
{                               //if they are synchronized
    if(observer.synchronized()) //let them read the data
     {
        observer.read();
     }
}
elapsedTime = elapsedTime + timeStep;
elapsedStep = elapsedStep+1;
```

As one can see, the sensors are updated firstly. Secondly, the controllers get their chance to exert influence on the simulation, this eventually leads to actuators changing torques and forces and thirdly the subsystems are called to actually execute one step of a physical simulation. The last step of the main loop is to call all synchronised observers, which update their representation of the simulation. The main loop itself is called by the public method `run` of `CIxSystemController`. This is where the loop occurs and this method has to be called from outside to run the simulation. Next, the functionality in pseudo-code:

```
while(runSimulation)
    {
    startTime = getTime();
    system.run();       // execute one step
    eventHandler.run(); // check if there are evens pending
    endTime = getTime();// if so execute them
    execTime = endTime - startTime;
    if(runRealtime)     // maybe we were faster than real-time
```

```
        {
            idle(execTime - timestep);
        }
    }
```

The event handler which is executed after the call of `run` will be described in the next section.

2.5 Parallelization and Event handling

Some actions which occur during the execution of an ibex-simulation have to be synchronised with the main loop. Otherwise the simulation integrity might be corrupted. Consider the following example: An external control unit wants to set a torque for a specific actuator. It must be guaranteed that this action occurs before or after the next timestep to be simulated. Otherwise the state of the simulation can be corrupted since its not possible to predict when exactly the effect will take place. Table 3 lists the different types of asynchronous events, which have to be caught and handled synchronously with the main loop.

Table 3. The different Asynchronous Events which can occur

Asynchronous Event Types
Actions taken by the user via the console or the GUI
Actions taken by external control hardware or software that is connected to Ibex
Actions taken by asynchronous observers (3d-engines)
Actions taken by a multi-threaded program that uses the ibex-framework.

To solve this problem in a generic way, Ibex provides its own event handler. It is called after a specified number of loops of the simulation have been executed. The event handler has a FIFO queue containing all pending events and cannot be interrupted once `handleEvent()` has been called. At the moment, every event has the same priority. For the future it is intended to implement different priorities for events. This would allow to distinguish between events which are handled every timestep and events that are only handled during idle time or after a given number of steps.

When a possibly dangerous action is issued by a application programm, the corresponding event is generated automatically. The application programmers do not have to specify events by themselves if they do not explicitly want to. Naturally, asynchronous events can still occur. A typical example is the event handler itself. When the events are handled, new events cannot be added to the list since this would destroy the used data structures. Therefore, Ibex offers the possibility to guard so called critical sections by mutual exclusion using the class `CIxMutex`. The next section describes how external hardware can be connected to Ibex making explicit use of multi-threading, the event handler and mutual exclusion.

2.6 Interfacing External Hardware

For HILS tasks Ibex must provide means to connect external hardware to the simulation. Since the number of possible connection technologies is nearly infinite, Ibex does not offer specific predefined protocols beside a standard TCP/IP connection. Ibex rather encourages the connection strategy presented in fig. 3.

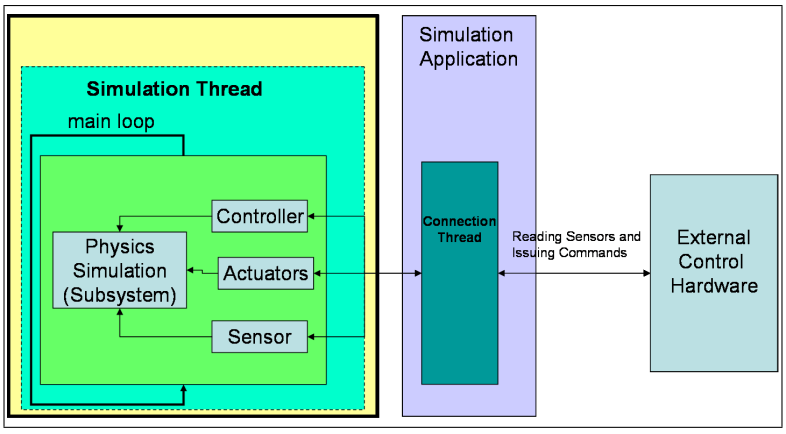

Figure 3. Connecting external hardware using a special connection thread

The basic idea is that the concrete applications using ibex create special subclasses of `AIxThread` which are responsible for the connection between the hardware and the simulation. Those threads acquire the data from the sensors and provide them in the required format to the external system. In the other direction, commands issued by the controller are translated by the connection-threads into a format suitable for ibex. To read data the thread can directly access the objects it is interested in. For the commands being executed, the situation is more complicated. Since they will certainly influence the simulation state, they have to be synchronised with the main loop. This can be achieved either by defining specific events or by extending `AIxActuator` with special methods guaranteeing mutual exclusion.

2.7 Graphical User interface

Ibex provides a standard graphical user interface (GUI) basing on the open source wxWidgets framework (wxWidgets 2005). Application using this GUI benefit from the already implemented standard functionality listed in tab. 4. This standard GUI is very quickly adapted to the needs of a specific application. An example for a typical GUI derived from the standard implementation can be seen in the section 4.2about the example elevator application (fig. 5).

Table 4. The standard GUI functionality

Functionality
Loading configurations from file.
Controls to start and stop a simulation and to run it with different speed options and/or for a given length of time.
Possibility to choose between 3 different 3D-graphics engines to visualize the simulation. In the 3D-Visualisation.
Visualisation of traces, collision points, joints and collision shapes in both textured or wireframe mode.
To allow a better view on certain objects in the scene, every object can be made invisible.
Predefined observers to log data like speed, energy and position of every object into a tabular data-structure for later analysis with external tools like Excel or Matlab.

3. Toolchain

One of the most important features of a good simulation software is the so called toolchain. The issue is to find an efficient way to get the configuration data of a simulation into the programm. This data usually consists of the object's geometry simulated in the different subsystems and the properties of the controllers and actuators interacting with those objects. Ibex provides the possibility to read in geometry data and configuration data for motors and controllers from a special XML (W3C 2004) data format. As an example, the following code snippet shows the definition of an object with a box collision-shape.

```
<RigidObject name = "box"  mass = "1f"   density = "0.0f" static =
"false"  visualisationMeshFile = "block.obj">
    <position x = "0.0f" y = "0.0f" z = "0.0f" />
    <orientation x = "0f" y = "0f" z = "0f" w = "1f" />
    <BoxShape x = "1.0f" y = "1.0f" z = "1.0f" materialIndex = "2" >
        <color r = "1f" g = "0f" b = "0.0f" a = "0.0f"  />
        <position x = "0.0f" y = "0.0f" z = "0.0f" />
        <orientation x = "0.0f" y = "0.0f" z = "0.0f" w = "1f" />
    </BoxShape>
</RigidObject>
```

Often the geometry of parts to be simulated already exists in the form of CAD data. A pluggin for the popular Solidworks CAD Software allows to enrich existing models with the necessary collision geometry and exports them in the XML format readable by Ibex. This dramatically reduces the time necessary to load new mechanical parts into the simulation and allows a truly incremental test and development process.

4. Application Examples

At the beginning of this section, two application examples and their special characteristics will be presented in more detail. After that, other fields of application are briefly outlined and plans for the future of ibex are revealed.

4.1 Delta Robot Simulator

The aim of this project has been the testing of the controller of the popular micro delta robot developed by Clavel (Clavel 1988). Figure 4 shows the robot as it is displayed in a 3d-graphics observer during the simulation.

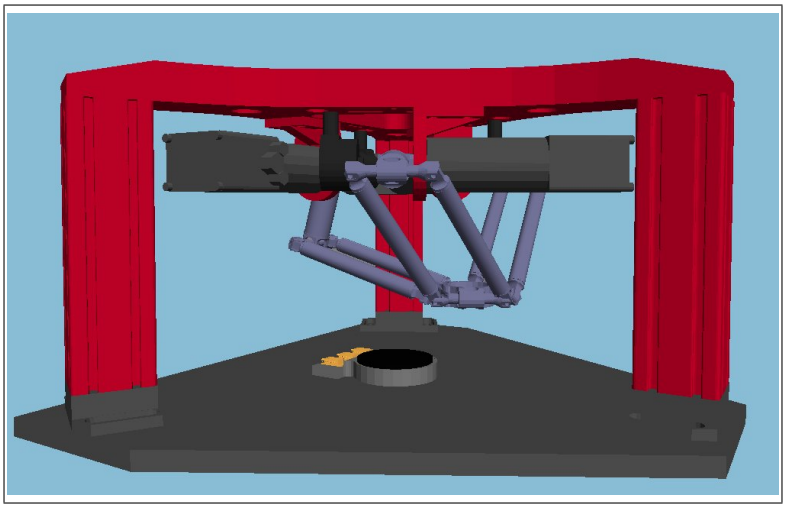

Figure 4. The micro delta robot during a simulation

The project has proven to be quite challenging because of two reasons. First of all, delta robots are extremely fast. The communication frequency between the controller and the robot is 2000 kHz. Therefore, the simulation must be able to run with a timestep of 0.0005 μs. The second problem is the closed kinematic loop that is generated by the robots parallel geometry (see fig. 4). This loop has been a problem for the fixed step rigid body dynamics engine used to simulate the robot. The solver had to be configured very carefully in order not to render the calculation of the position of the joints numerically unstable. The geometry of the robot bases on the existing CAD and has been created and loaded into the simulation using the toolchain presented in section 3. The communication between the real robot and the controller passes over TCP/IP. This connection has been replicated in the simulation. As described in section 1 this makes it completely transparent to the controller that the robot is only simulated. To interact with the environment the controller reads the position data of the encoders and feeds back the desired torques on the motors. Within the simulation these torques are then applied to the virtual actuators, which are connected to the upper arms of the robot.

4.2 Elevator Simulator

In this application the goal was the simplification of final tests done with standard elevator control units. These tests are compulsory before the control units are deployed and installed on their sites. At the moment, such tests are done with expensive mechanical simulators. A software simulation clearly has a big cost-cutting effect. The tests must be executed for the concrete type of elevator the control unit will steer. Therefore, the simulation starts with the acquisition of the configuration data of the elevator either from XML-file or through a GUI. The configuration data includes the number of floors, the number of doors, their sizes and types and other parameters. This information suffices to create a faithful representation of all the mechanical parts of the elevator. With the help of a flexible GUI the person executing the tests can then start them and overview the data exchanged between the controller and the software.

Figure 5 displays the simulation programm including its GUI. The 3D view in fig. 5 shows a door seen from inside the shaft. The simulation runs with up to 500 Hz and integrates a great number of totally different mechanical components. This also includes flexible structures like ropes and toothbelts, which cannot be simulated in real-time yet. Therefore, rigid body abstractions have been modeled and integrated into the simulation. For example, the rope is simulated as two springs connected to car and counterweight respectively. Since the aim of the simulation is not to test the behaviour of the rope but the control unit, such a simplification is valid. Another complication has been the different dimensions of the mechanical parts. While the cabin weights several hundred kilograms, the closing mechanism of the door consists of a number of very light, tiny components. This again can lead to numerical stability problems.

4.3 Other Applications and Future Plans

The Ibex framework has turned out to be extremely flexible concerning the possible applications. For example, earlier versions already have been successfully tested as tools for engineering education (Ettlin et al. 2005a). One focus of the ongoing development will be in the area of robot motion planning as it has been described by Ettlin et al. (2005b). Another target is to integrate Ibex into the popular Simulink software (Mathworks 2005) and thus making it even more easy to use in mechatronics development.

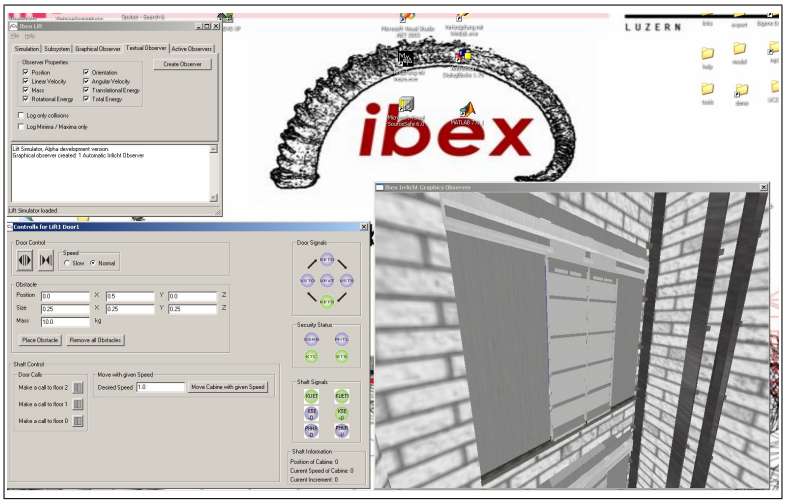

Figure 5. The elevator simulator during a simulation

5. Conclusion

Hardware in the loop simulation greatly simplifies testing and developing control hardware in robotics and mechatronics. The Ibex real time simulation frameworks offers the possibility to quickly set up simulations which exceed simple state machine based approaches. Powerful rigid body dynamics engines can be integrated and allow the simulation of complex mechanical structures like closed kinematic loops. Through its parallel architecture Ibex offers great performance. The state of a simulation can be visualised by so called observers in appealing 3d-graphics or it can be logged into predefined data-structures ready to statistical analysis. The use of Ibex has been proven by the example of a delta robot simulation and a complete elevator simulation. Ibex is a very flexible tool, it will be further developed to serve as a tool for research in the area of motion planning.

6. References

Baraff, D.,1989, "Analytical Methods for Dynamic Simulation of Nonpenetrating Rigid Bodies", Computer Graphics, 23, 3, pp. 223-232.

Baraff, D., 1991, "Coping with Friction for Non-penetrating Rigid Body Simulation", Computer Graphics, 25, 4, pp. 31-40.

Baraff, D., 1996, "Linear-Time Dynamics using Lagrange Multipliers", Proceedings of the 23rd annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH, pp. 137-146.

Clavel, R., 1988, "Delta, a fast robot with parallel geometry," Proceedings of the 18th International Symposium on Industrial Robots, pp. 91-100.

Ettlin, A., Büchler P., Trzebiatowski R., Vuillemin R., Bleuler H., 2005, "Real-Time Physics Simulation In Education", Proceedings of the Global Congress on Engineering and Technology Education GCETE 2005.

Ettlin, A., Büchler P., Bleuler H., 2005, "A Simulation Environment for Robot Motion Planning", Fifth International Workshop on Robot Motion and Control RoMoCo.

Gamma E., Helm R., Johnson R., Vlissides J., 1995,"Design Patterns", Addion Wesley, New York, United States.

Gebhardt, N. , 2005, "The Irrlicht 3D-Graphics Engine", http://irrlicht.sourceforge.net.

Smith, R., 2005, "The Open Dynamics Engine", http://www.ode.org.

Mirtich, B., 2000, "Timewarp rigid-body simulation", Proceedings of ACM SIGGRAPH 2000, pp. 193-200.

AGEIA Technologies Inc., 2005, "The Novodex Physics SDK", http://www.novodex.com.

Solidworks Corporation, 2005, "The Solidworks CAD Software", http://www.solidworks.com.

The Mathworks, 2005,"Simulink - Simulation and Model-Based Design", http://www.http://www.mathworks.com/products/simulink/.

The World Wide Web Consortium, 2004,"Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation 04, http://www.w3.org/TR/2004/REC-xml-20040204/.

The wxWidgets Team, 2005,"The wxWidgets cross-plattform native UI-Framework", http://www.wxWindows.org.