A Simulator for ACME
Term Project, CPSC 515
Final Report

Chris Majewski Shingo Takagi Tim Wellhausen

May 3, 1999

Abstract

The ACME[1] facility is a collection of robotic and measurement devices, programmable via a Java API, for modeling the characteristics of physical objects. We present a simulator of the ACME devices, integrated transparently with the existing ACME software. One application for such a simulator is as a testing ground for experiments, allowing them to be previewed and debugged before deployment on real (and expensive) hardware.

1 Introduction

The ACME software enables the development of experiments in the form of *user programs* containing *motion plans*, and ultimately controls the hardware by native calls to RCCL[2]. Our simulation is implemented by "cutting" just above the RCCL level, replacing the native calls with Java code. A user program is made to run as a simulation, rather than on the real hardware, by adding one line:

```
configManager.useSimulation();
```

The simulation takes the form of a 3-D animation, displayed in a *viewer* containing *scene graphs* corresponding to the various ACME devices. Collisions are detected and reported to the user.

Our implementation currently provides a kinematic model [1, 3] of four robotic devices mounted on a table. Extension to a dynamic model and support for additional devices is feasible within the framework we have chosen. Adding new

[1] ACtive MEasurement

devices – given support by the rest of the ACME package – would require a minimal amount code (the hardest part probably being the creation of a new scene graph). Moreover, by closely following the ACME design paradigm, we have made the simulator compatible with the (seemingly imminent) extension of ACME to a multi-user client-server architecture.

The simulator software can be logically subdivided into an *API*, which is the ACME programmer's interface to the simulation; a *viewer*, which displays the simulation and detects collisions; and a *back–end*, which performs kinematic[2] computations and updates the viewer.

The remainder of this report is organized as follows:

- Section 2 gives an overview of the design.
- Section 3 discusses the implementation details of the simulator API and back–end.
- The viewer is discussed in section 4.
- Section 5 suggests further work on the simulator.
- Section 6 gives a pointer to the documentation.
- The contributions of the individual team members are outlined in section 7.

2 Design

As an experimental project, ACME is under constant development. This means that even the current design is subject to change. As a result, we decided to implement the simulator at a low level, placing a minimum of constraints on the further development of ACME. It should be possible to use our simulation even in a changed ACME environment.

Our goal was to simulate the RCCL connection to the real robots. ACME distinguishes internally between a RCCL server and a RCCL client. In simulation mode, the two RCCL connection objects (client and server) are replaced by a simulation client/server. Since the simulation classes implement the same interface as the original RCCL classes, a simulation is programmatically transparent to the end user.

The simulation server dispatches motions according to their target device (e.g. Puma robot or Linear Stage). Each device corresponds to a DeviceSimulation subclass. Objects of this class perform the necessary kinematic computations on each motion, then pass the processed motion to the viewer. The viewer

[2] Eventually, dynamic

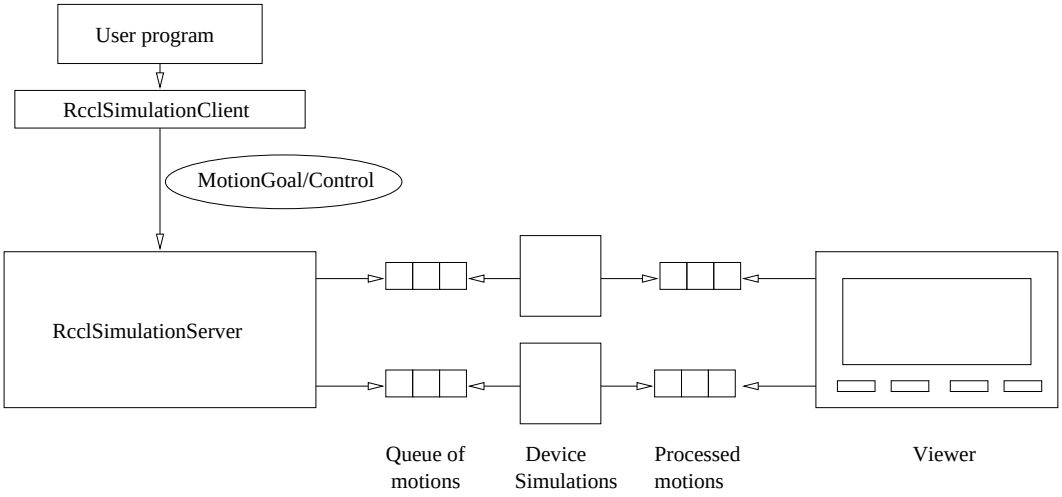

Figure 1: Architecture of the ACME simulation

object extracts the necessary information (currently, the desired joint values) and hands it to the appropriate scene graph object for rendering and collision detection.

3 Implementation

The simulation code resides in a new package, `acme.simulator`. The viewer lives in a separate package, `acme.simviewer`. It is discussed in detail in section 4.

Very few changes were made to the original ACME code in order to integrate it with the simulation. The ConfigManager class received a new method, `useSimulation()`, which enables the simulation by creating an RcclSimulationServer object. This method has to be called before any device is initialized. To allow the connection to the simulation, two more classes needed slight changes. The first is RcclActuator, which originally connected only to the RCCL server. The second is kinematics.GenericKinematics, which, in simulation mode, uses simulation.RobotInfo instead of server.RobotInfoServer.

3.1 Simulator API

The simulator API consists of a server class (RcclSimulationServer), a client class (RcclSimulationClient), and a class which provides static device parameters (RobotInfo).

RcclSimulationServer and RcclSimulationClient

The simulation server drives the simulation. It implements the RcclConnection interface, providing all the high-level robot control functionality specified by this interface. When a user program is run in simulation mode, the connection to RcclConnectionServer is replaced by a connection to RcclSimulationServer. This connection is established by RcclSimulationClient, a thin layer analogous to RcclConnectionClient. An instance of RcclSimulationClient is created for each robotic device needed by a user program.

A simulation is initiated by a call to the new `useSimulation()` method in ConfigManager. This creates a new RcclSimulationServer object. The server in turn instantiates the viewer, which opens a window and displays the ACME table with its various devices.

ACME user programs specify the devices they need by registering them with a Platform object. For each device, the Platform object creates a new instance of the appropriate RcclActuator subclass. It then calls the ConfigManager's `Install()` method on that instance. The ConfigManager initializes each device by calling RcclActuator's `initialize()` method. Normally, this method opens a connection to RCCL by creating a new RcclConnectionClient instance. In simulation mode, however, the RcclConnectionClient is replaced with a RcclSimulationClient.

Since both RcclConnectionClient and RcclSimulationClient implement the same Connection interface, the `open()` method of this interface can then be (and, in fact, is) called to complete device initialization. In simulation mode, the `open()` method calls a corresponding method in RcclSimulationServer. This creates a DeviceSimulation instance of the proper subclass (e.g., the Test Station device corresponds a TestStationSimulation instance). The following example code shows how a typical user program might initialize its devices:

```
Platform platform = new Platform (false, false);
ConfigManager configManager = platform.getConfigManager();

configManager.useSimulation (); // now in simulation mode
platform.Install (DeviceId.TEST_STATION);
platform.Install (DeviceId.CMS); // Puma robot
platform.Install (DeviceId.FMS); // Gantry device
platform.Install (DeviceId.LINEAR_STAGE);
configManager.initialize();
```

RobotInfo

The simulator's RobotInfo object implements ACME's Info interface to provide data about the static parameters of each device. For example, the number of

joints in the Puma robot named "thumper" can be obtained by calling RobotInfo's `getNumJnts()` method with the argument "thumper".

These parameters are stored in text files with the extension `.jls`. The RobotInfo object keeps a JLSFile object for each device. The JLSFile class parses a .jls file when it is instantiated, and stores the appropriate parameters for use by RobotInfo[3].

3.2 Simulator Back–end

The simulator back-end consists of DeviceSimulation subclasses and a modified simulator.Motion class.

DeviceSimulation

The DeviceSimulation object resolves device motions into joint motions. Recall that the RcclSimulationServer's `open()` method is called to initialize a device. This method creates a new instance of the appropriate DeviceSimulation subclass. Here is the definition of a typical subclass:

```
package acme.simulator;

import acme.Kinematics;
import acme.DeviceId;
import acme.kinematics.StageKinematics;

public class TestStationSimulation extends DeviceSimulation {

  public TestStationSimulation () {
    super ("stage");
  }

  public int
  getDeviceId()
  {
    return DeviceId.TEST_STATION;
  }

  protected Kinematics
  getKinematics()
  {
```

[3]The .jls files are read from the acme.simulator directory. This is done to avoid any dependencies on the RCCL installation. Thus, any modifications to the original description files must be copied to the acme.simulator directory if they are to have any effect on the simulation.

```
    if (kinematics == null) {
      return new StageKinematics(deviceName);
    }
    return kinematics;
  }
}
```

Since devices may move simultaneously, each DeviceSimulation runs in its own thread. This thread takes motion blocks from a queue and processes them using a Kinematics subclass. When the queue is empty, the thread blocks until RcclSimulationServer enqueues another motion block. The thread exits when the user program uninstalls the device.

After it creates a DeviceSimulation instance, the simulation server registers the viewer as a listener for that instance. Since the DeviceSimulation keeps a vector of listeners, it is possible, in principle, to have any number of viewers displaying a single simulation.

Once the kinematics of a motion have been resolved, it is time-stamped and marked as RUNNING. The DeviceSimulation instance then notifies its listeners, handing them the processed motion. When the viewer is notified, it updates the scene. The viewer signals motion completion via the DeviceSimulation `doneMotion()` callback, which puts another time-stamp on the motion, and marks it COMPLETED.

Motions

ACME's Motion objects are high-level descriptions of a given device motion. As such, they reference a MotionGoal, a MotionCtrl, and a target device. The MotionGoal specifies the geometry of the motion, that is, a set of joint values in the case of a JOINT_GOAL, or a homogeneous transformation matrix in the case of a CARTESIAN_GOAL. The MotionCtrl object specifies how the desired goal is to be achieved – essentially, the dynamics of the motion. The target device corresponds to the robot which is to carry out the motion.

A set of successive motions can be encapsulated in a MotionPlan. The MotionPlan's `execute()` method resolves each motions into its MotionGoal, MotionCtrl, and target device components. These components are passed to the RcclSimulationServer, which reassembles them into instances of our new simulator.Motion class.

The RcclSimulationServer keeps a separate *motion block* (queue of motions) for each target device. Each new simulator.Motion object is then placed in the appropriate motion block. Once all motions have been dispatched by the MotionPlan, each motion block is placed on the input queue of the corresponding DeviceSimulation subclass.

Note that simulator.Motion objects are only used by the simulator internally. Their purpose is to provide the data required by Connection-level functions like `getCompletionTime()`. They are also used to exchange motion information between the simulator and the viewer. The original acme.Motion object is still the only one visible to the end user.

The ACME package also provides a Motion subclass called Move, which facilitates the definition of typical robot motions. The following example shows motions at work.

```
Fms gantry = (Fms) Platform.getDevice(DeviceId.FMS);
Cms puma = (Cms) platform.getDevice(DeviceId.CMS);
double[] jointValues = {2.0, 1.0, 1.0, -(Math.PI/4.0), Math.PI/2.0};
Matrix4d matrix = new Matrix4d();
// initialize matrix ...
Move pumaMove = new Move(puma,
 matrix,
 acme.kinematics.PumaKinematics.RIGHT_CONF);
Move gantryMove = new Move (gantry, jointValues);

plan.add(pumaMove);
plan.add(gantryMove);
plan.execute();
```

Device state

When a DeviceSimulation instance is created, it obtains the initial state of each device from RobotInfo. Each initial state is defined by the "parkAngles" specified in the relevant .jls file. A simulation starts with the devices in their "parked" states. For example, the parked state for the Puma robot is given by:

```
# (NJ floats)    parkAngles       the park angles for the robot
0.0
90.0
-180.0
180.0
90.0
90.0
```

We assume that joint values in user programs will be in the same coordinates as those in the .jls files, i.e. that they are not relative to the parked states. The viewer takes initial states to be $[0, 0, \ldots, 0]$, but what it displays are the parked states. We therefore calibrate by subtracting the parked values when the viewer processes a motion.

Device states may be obtained at any time from the ACME DeviceState subclasses, as they would in the course of an experiment on the real hardware. The difference is that in simulation mode, the DeviceState subclasses get the data from the simulation server instead of the RcclConnectionServer. The server gets the state data from a DeviceSimulation instance.

4 Viewer

Please see `http://www.cs.ubc.ca/spider/stakagi` for the contents of this section.

5 Looking ahead

The first logical extension to our work would probably be support for motion dynamics. Some additional features might be:

- An autonomous viewer, which is started once and then runs as a daemon which can display successive executions of a user program.
- Support for persistent device state. This could be done by writing joint angles to a file, then reading them back in when a new user program is run.
- Interaction between the viewer and the ACME hardware. For example, the viewer could be used as a remote client which displays a live user experiment taking place in the ACME lab. Conversely, the viewer might be used as a WYSIWYG interface for manipulating the hardware. (This would have to be implemented very carefully!)

6 Documentation

A copy of the javadoc API documentation is in `file:///lci/project/acme/majewski/javadoc/index.html`. The relevant packages are acme.simulator and acme.simviewer. A sample user program is acme.simulator.Test.

7 Contributors

Peering fearlessly beneath thorny layers of abstraction in the ACME package, Tim Wellhausen decyphered its inner workings and proposed the original design for the simulator. The simulator API, as well as classes for reading device parameters, are his work.

Shingo Takagi is the man behind the viewer (package acme.simviewer). He is responsible for the spectacular and dizzyingly realistic animation you see when a simulation is run.

Chris Majewski is responsible for the DeviceSimulation and related classes, the interface between the simulator and viewer, debugging, testing, and last-minute fixxes.

8 Acknowledgements

We would like to thank Jochen Lang for introducing us to the ACME facility.

References

[1] C.S. George Lee. Robot arm kinematics, dynamics, and control. *Computer*, pages 62–80, December 1982.

[2] John Lloyd and Vincent Hayward. Multi–RCCL user's guide. Software manual, Computer Science Department, University of British Columbia, 201-2366 Main Mall, UBC, Vancouver, B.C. V6T 1Z4, April 1992.

[3] Richard P. Paul. *Robot manipulators.* MIT Press, Cambridge, Massachusetts, 1981.