Modeling an AERCam: A case study in modeling with concurrent constraint languages

Lars Ålenius* Vineet Gupta†

October 5, 1998

Abstract

This paper presents a model for the Sprint AERCam, a beachball sized robotic camera that floats around in the cargo bay of the Space Shuttle, allowing the astronauts and ground mission control to have an additional view of the task they are trying to accomplish. It has a self contained propulsion system, giving it the capability to maneuver with six degrees of freedom.

We present a model for the AERCam written in Hybrid cc, an extension of cc programming for modeling continuous/discrete systems. We modeled both the dynamics and control of the AERCam in Hybrid cc, and interfaced this model with an animation interface which allows a user to interact with the model in real-time.

1 Introduction

This paper presents a model for the Sprint AERCam, an Autonomous EVA Robotic Camera[Phi97]. The AERCam is a beachball sized camera (see Figure 1), that floats around in the shuttle bay, allowing the astronauts inside and outside the shuttle and ground mission control to have an additional view of the task they are trying to accomplish. It has a self contained propulsion system, giving it the capability to maneuver with six degrees of freedom (three rotational and three translational). The AERCam has two cameras which relay a video stream to a laptop held by the crew members doing extravehicular activity, or to the ground. It can be controlled by sending commands from a small station in the shuttle. The AERCam is one of the key technologies required for the International Space Station — future versions will also be able to do repairs in addition to taking pictures.

In this paper we present a dynamical model of the AERCam written in Hybrid cc, an extension of cc suitable for modeling physical systems. We modeled the dynamics of the AERCam, and also developed a simple controller in Hybrid cc for controlling it. We then hooked up this model with World ToolKit ®, providing a user the capability of interacting with the AERCam in real time—both issuing commands and receiving the video stream are modeled.

The use of a constraint based language was particularly helpful in modeling the dynamics as it allowed us to write the differential equations directly, letting the Hybrid cc interpreter figure out the proper way to simulate these equations. Concurrency was crucial for structuring the model — we just wrote the dynamics model and the controllers for the different axes and allowed them to run concurrently; their interactions were managed automatically by the ask-tell synchronization of cc. While the ability of Hybrid cc to solve

*Kungl Tekniska Högskolan, SE-100 44 Stockholm, `t93_als@t.kth.se`

†Caelum Research Corporation, NASA Ames Research Center, M/S 269-2, Bldg 269, Rm 127, Moffett Field CA 94035, `vgupta@ptolemy.arc.nasa.gov`

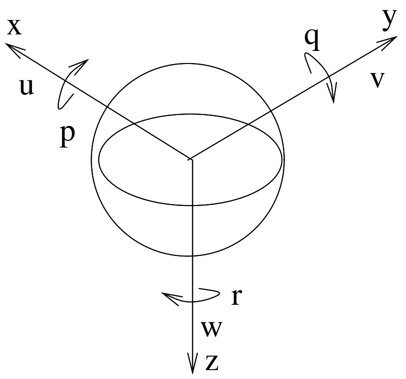
Figure 1: The Sprint AERCam

constraints was not significantly exploited in this system, we expect it to be used a lot more when the dynamics are more complicated.

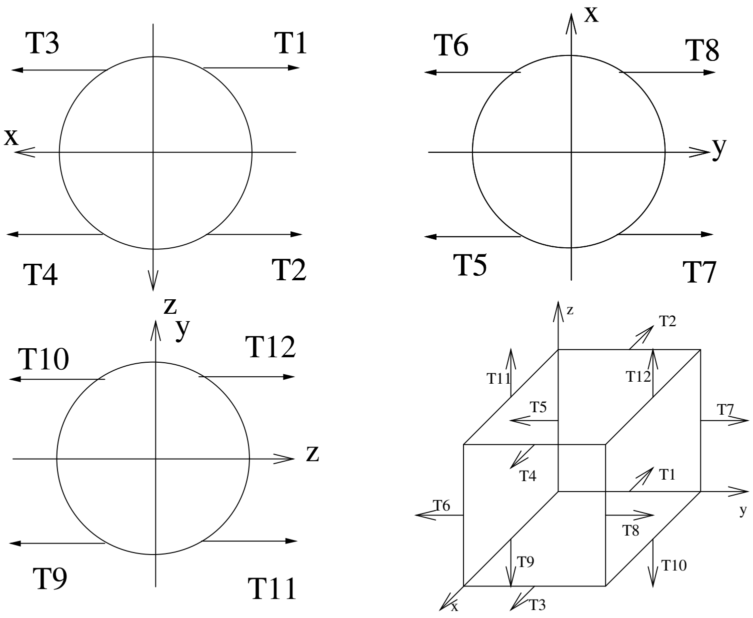
A model like this could potentially be put to many different uses:

- It provides a simulation testbed for testing control algorithms for the AERCam.
- It allows a ground crew to play the entire sequence of operations of a mission, making sure that there is enough fuel, and that all the objectives of the mission will be met. In fact a video stream of what might be seen can be generated.
- It can be used for planning the command sequence of the AERCam. This is important in an environment like a Space Station, where there may be many obstacles in the path of the AERCam. The planner can then optimize on the time and the fuel spent.
- It can be used to develop algorithms for diagnosis of the AERCam, in case a failure occurs.

In the next section, we provide an overview of the dynamics and control of the AERCam. We then describe how the dynamics were modeled in Hybrid cc, showing the correspondence between the Hybrid cc code and the physics equations. We then show some representative samples of code implementing the controllers. Finally we show how the simulation was hooked up with the animation, and discuss the performance of the system.



The Body frame of reference and the directions of velocities (u,v,w) are the components of the translation velocity, while (p,q,r) are components of the angular velocity.



Three views of the AERCam, showing the thrusters, and showing all the thrusters together in the cube circumscribing the AERCam.

Figure 2: The AERCam axes and thrusters

2 The AERCam system

The AERCam is a small spherical robotic camera unit, with 12 thrusters for manipulation (Figure 2). It weighs about $15.33kg$, and has a radius of $14cm$. It also carries $0.544kg$ fuel. For the purposes of this model, we assumed the sphere to be uniform, and the fuel to be in the center. The fuel is depleted as the thrusters fire, so we simply assumed that the mass of the AERCam decreased uniformly as the fuel was spent. This was done to simplify the physical equations, not to help the simulation.

The dynamics of the AERCam are described with reference to the AERCam body frame of reference. The translation velocity of this frame with respect to the shuttle inertial frame of reference is 0, however its orientation is the same as the orientation of the AERCam — thus its orientation with respect to the shuttle reference frame changes as the AERCam rotates (it is not an inertial frame). There are twelve thrusters, four along each major axis in the AERCam body frame, as shown in the figure. The positions of the thrusters can be imagined to be on the centers of the edges of a cube circumscribing the AERCam. Thrusters T_1, T_2, T_3, T_4 are parallel to the X-axis and are used for translation along the X-axis or rotation around the Y-axis. Thus we can fire thrusters T_1 and T_2 to get translation along the positive X-axis, and thrusters T_1 and T_4 to get a negative rotation around the Y-axis. Similarly, thrusters T_5, T_6, T_7, T_8 are parallel to the Y-axis, and are used to rotate around the Z-axis, and thrusters $T_9, T_{10}, T_{11}, T_{12}$ are parallel to the Z-axis, and are used for rotation around the X-axis. Since the same thrusters are used for rotation and translation, the AERCam is either translating or rotating. Note that each thruster can be either on or off, thus its behavior is discrete.

For safety of the crew and the shuttle equipment, the maximum velocity of the AERCam is set to 7.62 cm/s. Also the maximum angular velocity is set to 0.5236 rad/s.

2.1 AERCam dynamics

Quaternions. The orientation of the AERCam is modeled as a *quaternion* (q_1, q_2, q_3, q_4), with $\sum_{i=1}^{4} q_i^2 = 1$. Quaternions are based upon Euler's theorem on Rotations:

> The attitude of a body with respect to a given frame can be described as a rotation about some axis.

Thus if the direction of the axis of rotation is given by (n_1, n_2, n_3), and the angle of rotation around it is ϕ, then we have $q_i = n_i sin(\phi/2)$, for $i = 1, 2, 3$ and $q_4 = cos(\phi/2)$. A quaternion can be written as $q = q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} + q_4$, where $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1, \mathbf{ij} = -\mathbf{ji} = \mathbf{k}, \mathbf{jk} = -\mathbf{kj} = \mathbf{i}, \mathbf{ki} = -\mathbf{ik} = \mathbf{j}$. Rotations can now be composed by multiplying quaternions. Quaternions are particularly useful for simulation, as numerical errors can easily be reduced by normalizing.

We model a simplified version of the dynamics of the AERCam, based on Newtonion laws[ER95]. The position of the AERCam is modeled as a triple (x, y, z) in a coordinate system whose axes were fixed to the space shuttle (the inertial frame I). The the complete position and orientation of the AERCam is a 7-tuple $(x, y, z, q_1, q_2, q_3, q_4)$. The translation dynamics can be derived as follows. Let $\vec{\mathbf{V}}$ be the velocity in the AERCam body frame, with its vector components given by (u, v, w). The frame rotates with respect to the inertial reference frame with velocity ω, which is the same as the angular velocity of the AERCam — its components are given by (p, q, r). Since the Body frame is a rotating frame, we need to add the Coriolis force to the forces acting upon the AERCam. We are assuming uniform velocity of rotation (since while the AERCam in translating it is not rotating)[Arn78, pg. 130].

$d(m\vec{\mathbf{V}})/dt = \vec{\mathbf{F}} + 2m(\vec{\mathbf{V}} \times \vec{\omega})$ Newton's Law

$\vec{\mathbf{V}}\, dm/dt + md(\vec{\mathbf{V}})/dt = \vec{\mathbf{F}} - 2m(\vec{\omega} \times \vec{\mathbf{V}})$

Now we can expand this out and write down one equation for each coordinate.

$$
\begin{aligned}
du/dt &= F_x/m - 2(qw - vr) - (u/m) * dm/dt \\
dv/dt &= F_y/m - 2(ru - pw) - (v/m) * dm/dt \\
dw/dt &= F_z/m - 2(pv - qu) - (w/m) * dm/dt
\end{aligned}
$$

Table 1: The translational dynamics equations

For the rotational dynamics, we start with the equation $d(\mathbf{I}.\,\vec{\omega})/dt = \vec{\mathbf{M}}$, where $\mathbf{I}$ is the inertia tensor, $\vec{\omega}$ is the angular velocity in the Body frame and $\vec{\mathbf{M}}$ is the total moment on the AERCam. The calculations are greatly simplified by the assumption that the AERCam is a uniform sphere — thus $\mathbf{I}$ is the product of a scalar I with the identity matrix.

$d(\mathbf{I}.\,\vec{\omega})/dt = \vec{\mathbf{M}}$

$d(\mathbf{I}.\,\vec{\omega})/dt = dI/dt\,\vec{\omega} + Id(\vec{\omega})/dt$

Now we can substitute $\vec{\omega} = (p, q, r)$ into the second equation and write it out coordinate wise, giving us the rotational dynamics.

2.2 Controlling the AERCam

We implemented two modes of controlling the AERCam — Position control mode and Velocity control mode. The controllers for each of these were slightly different.

Position control mode. In this mode we command the AERCam to go to a particular point and point the camera towards another specific point. The procedure for this is as follows:

- Rotate the AERCam so that one of its body frame axes is parallel to the desired direction of motion (to the target point).
- Fire thrusters along the axes to move the AERCam towards the target point. Rotating first allows us to fire one set of thrusters, this saves fuel.
- When the AERCam is close to the target, fire thrusters to make corrections, so that the exact target point is reached. Slow it down to a complete stop when that happens.
- Rotate the AERCam to point it towards the desired point

We have one controller for each axes of rotation and translation such that if the system receives a command from the above procedure to reach a particular position on that axis, it fires a series of thrusters to reach that position. The controller we have implemented is a Schmitt trigger [Bry94, pg 34]— this is a modification of a bang-bang controller with a linear switching function. The eventual goal of such a controller is to take v_{error} and d_{error} to 0. If the d_{error} is positive, it fires thrusters to decrease it. This increases v_{error}, so when $v_{error} = d_{error} * c$, the thrusters fire in the opposite direction, to decrease v_{error}, and so on, as shown in Figure 3. Careful selection of the switching constant c ensures that the error in both velocity and position keeps decreasing. Bang-bang controllers use a lot of fuel as they exhibit chattering around the desired point $v_{error} = d_{error} = 0$, so using a modified controller with a dead-band region, a bang-off-bang controller, allows the error to be reduced to some tolerance, without wasting fuel.

Velocity control mode. In this mode, the AERCam is commanded in real-time by a user with a joystick like object. The thrusters fire to reach the velocity specified by the user.

As for the position control mode, we have one controller for each axes of rotation and translation such that if the system receives a command from the user to reach a particular velocity on that axis, it fires a series of thrusters to reach that velocity. The controller we implemented was a conventional bang-off-bang controller — thus if the velocity error exceeded v_{high} it fired a set of thrusters to correct it, and if the error became less than v_{low}, then the thrusters were switched off. In order to minimize the firing of thrusters, the given velocity was discretized to 11 levels, this was necessary to prevent the thrusters from firing all the time due to miniscule changes in the commanded velocity.

3 The Hybrid cc programming language

Hybrid cc [GJS98, CG98] is an extension of cc, intended for modeling hybrid systems, which are systems with both discrete and continuous change. Our execution model for these systems is a piecewise continuous execution model. Thus a program execution is a sequence of alternating *point* and *interval* phases — discrete changes occur in the point phase while the system evolves continuously in an interval phase. In a point phase, we extract a cc program from the given Hybrid cc program. This program is executed to determine the behavior of the system at that instant. Then for the succeeding interval phase, based upon the point phase execution, we extract another cc program from the given Hybrid cc program, and execute it. Its output will contain some time varying constraints, like differential equations, which govern the behavior of the system in the interval phase. These differential equations are solved (numerically or analytically) to determine the values of the variables in the interval phase. The interval phase is terminated and a new point phase is entered when one of the guard conditions changes status — these guard conditions are all the constraints whose entailment was checked by the cc program at the beginning of the interval phase. Intuitively, the interval phase continues as long as all the asks in the interval cc program would have the same result, and hence the constraints in the store would be the same. After the point phase the system then enters another interval phase, and the process continues.

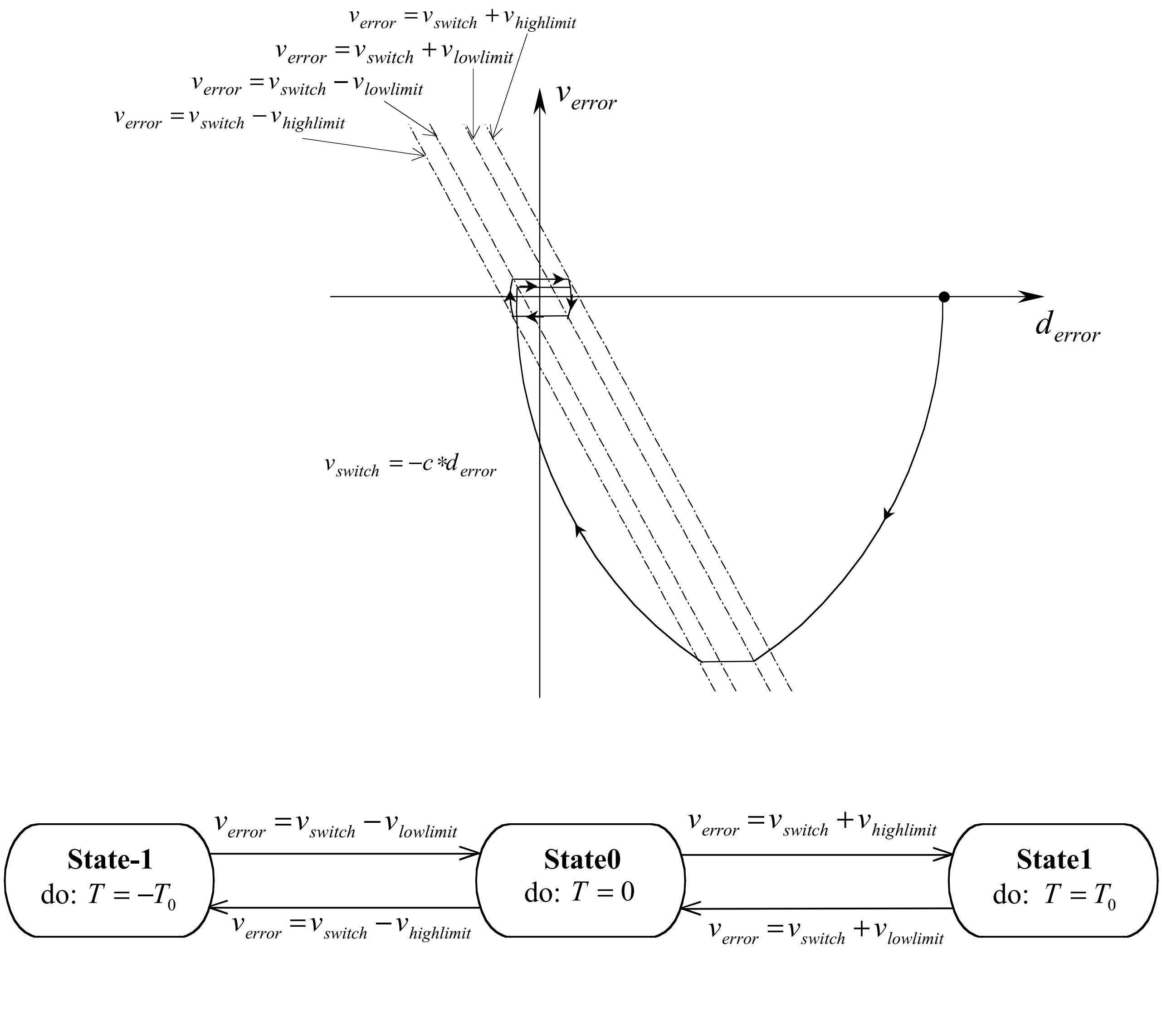


Figure 3: The Schmitt trigger. The top figure gives the change in v_{error} and d_{error} as the controller fires thrusters. Note the limit cycle towards the end, the controller is shut off when this is reached. The bottom figure shows the automaton.

We use the eventual tell version of cc [Sar93]. As usual, the store is represented as a constraint, which various independently acting programs can add constraints to or ask if some constraints are entailed. Programs are constructed using the standard combinators: c adds the constraint c to the store; **if** c **then** A asks the store if c is entailed, if it is then it reduces to A, otherwise it suspends; A, B reduces to the independent parallel execution of A and B; **new** X **in** A creates a new local variable which can be referred to only in A; $g(X_1, \dots X_n)$ reduces to $A(X_1, \dots X_n)$, where A is the code of the procedure associated with g (this is declared via constraints, see below).

To this basic set of combinators we add two new combinators. **if** c **else** A comes from default logic [Rei80] — it reduces to A if c is not entailed in the current phase, otherwise it does nothing. Note that this is a non-monotonic construct, so for A to be executed we require for determinism that c not be entailed anywhere during the execution of the current phase. This allows us to define strong preemption constructs as defined in [Ber93]. The other construct we add is **hence** A, which allows programs to extend across time. In a point phase, **hence** A does nothing, however it forces A to be executed in the cc program in the succeeding phase, and in every point and interval phase after that. In an interval phase, it forces A to be executed in each succeeding phase, including the current one.

Our constraint system consists of algebraic equalities and inequalities over interval-valued arithmetic variables and real constants. We have a number of built in arithmetic functions like exp, log, sin, cos etc. In addition we have nonnumeric constraints, these allow us to define string equalities, closures and classes (thus the procedure declarations in cc programs are now done via constraints). Using the basic set of combinators, we have implemented a number of other combinators, some of which will occur in our model. Examples are

- **always** A. This is like A, **hence** A, it executes A in every succeeding phase, including the current one.
- **if** c **then** A **else** B. This is like **if** c **then** A, **if** $\neg c$ **then** B.
- **when** c **do** A. This reduces to A in the first phase in which c is entailed.
- **do** A **watching** c. This behaves like A, except that in the first phase in which c is entailed it stops A and any programs A may have reduced to.
- **next** A. In an interval phase it does nothing. In a point phase, it forces the creation of another point phase immediately after the current one, and A is executed in that phase, along with any programs that were scheduled to execute by **hence** . This allows us to do a succession of discrete changes, effectively embedding Timed Default cc([SJG95]) inside Hybrid cc.

4 The dynamics of the AERCam

The dynamical behavior of the AERCam was modeled in the obvious way: we wrote down the equations described above, for each coordinate. The first 3 equations are the equations written in 1. The next three equations compute the total force along each axis. The next two sets of equations provide the rotational dynamics and compute the rotational torques, as described by the equations for rotational dynamics. The last two sets of equations describe the change in the position based on the velocities. In the body frame of reference we would have $x' = u$ etc., but here we have converted them to the inertial frame using quaternion transformation by matrix multiplication.

In this piece of code, `T1, ...T12` are the 12 thrusters. Body is another class defining the basic parameters of the AERCam like mass(`m`), moment of inertia(`in`) and distance of the thrusters from the center.

```
DynamicsClass =
 ()[u, v, w, p, q, r, x, y, z, q1, q2, q3, q4,
  x_force, y_force, z_force, l_moment, m_moment, n_moment]  {

  always {
    u':=(x_force)/Body.m-2*(q*w-r*v)-u*Body.m'/Body.m,
    v':=(y_force)/Body.m-2*(r*u-p*w)-v*Body.m'/Body.m,
    w':=(z_force)/Body.m-2*(p*v-q*u)-w*Body.m'/Body.m,

    x_force:=T1.thrust+T2.thrust-T3.thrust-T4.thrust,
    y_force:=T5.thrust+T6.thrust-T7.thrust-T8.thrust,
    z_force:=T9.thrust+T10.thrust-T11.thrust-T12.thrust,

    p':=l_moment/Body.in-p*Body.in'/Body.in,
    q':=m_moment/Body.in-q*Body.in'/Body.in,
    r':=n_moment/Body.in-r*Body.in'/Body.in,

    l_moment:=(-T9.thrust+T10.thrust+T11.thrust-T12.thrust)*Body.a,
    m_moment:=(-T1.thrust+T2.thrust+T3.thrust-T4.thrust)*Body.a,
    n_moment:=(-T5.thrust+T6.thrust+T7.thrust-T8.thrust)*Body.a,

    x':=u*(1-2*q2^2-2*q3^2)+v*2*(q1*q2-q3*q4)+w*2*(q3*q1+q2*q4),
    y':=u*2*(q1*q2+q3*q4)+v*(1-2*q3^2-2*q1^2)+w*2*(q2*q3-q1*q4),
    z':=u*2*(q3*q1-q2*q4)+v*2*(q2*q3+q1*q4)+w*(1-2*q1^2-2*q2^2),

    q1':=(q4*p-q3*q+q2*r)/2,
    q2':=(q3*p+q4*q-q1*r)/2,
    q3':=(-q2*p+q1*q+q4*r)/2,
    q4':=(-q1*p-q2*q-q3*r)/2
  }
}
```

The basic idea is that the equations are written exactly as they were computed, without much extra manipulation to make them suitable for execution. The only manipulation we did was to change the equality into "`:=`" — this is a Hybrid cc operator showing that we know that the left hand side has the dependent variables, and the right has the independent variables: this is useful for increasing the speed of execution, but not for the correctness.

The remaining classes describe the Body parameters and the thruster behavior. The first class describes the Body — it has four properties mass, inertia and fuel remaining and distance of the thrusters from the center. The `body_mass` is a constant, only the fuel decreases when the thrusters are fired. As the mass decreases the inertia also decreases. Note that from the equation $in = (2/5) * m * radius^2$ we infer $in' = (2/5) * m' * radius^2$ since $radius$ is a parameter and hence a constant.

The thruster class describes the behavior of a thruster. The controller sets the voltage level of the thruster, and if the voltage is positive and there is fuel, then thrust is set to `thrust_level` otherwise it is 0. The next statement is a default: it asserts that if no other agent has set the voltage level to something positive, then it must be 0. The advantage of this is that the controller does not have to switch the thruster off, it is always off unless switched on. In this model we did not represent delays in the firing of the thrusters, this will be incorporated in more sophisticated models.

```
BodyClass =
 (body_mass, radius, init_fuel, c, thruster_dist)
 [m, in, fuel, a]  {
  fuel = init_fuel,
  always {
    a := thruster_dist,
    m:=body_mass+fuel,
    in:=(2/5)*m*radius^2,
    -fuel'*c:=sum(T.thrust, ThrusterClass(T), 1=1)
  }
},

ThrusterClass =
 (thrust_level)
 [voltage, thrust]  {
  always {
    if (Body.fuel = 0 || voltage = 0) then thrust=0
    else thrust=thrust_level,
    unless (voltage > 0) then voltage = 0
  }
}
```

The AERCam dynamics is completed by creating the appropriate objects by instantiating each of these classes.

5 The AERCam controllers

We implemented two global controllers — one for the position control mode and one for the velocity control mode as described in section 2.2. Each controller also had 6 subcontrollers for the three degrees of rotation and the three degrees of translation — these were switched on and off by the respective global controllers. The switch between the position and velocity controllers was done by the user, as described in the next section. The controller code was the bulk of the system, so we will present only some of it here.

Velocity control. We first describe the six controllers controlling the three degrees of rotation and the three degrees of translation in the velocity control mode. The arguments to each such controller are the 4 thrusters it controls, the parameter it is controlling, the lower cutoff and the upper cutoff and the maximum value of the parameter. We first discretize the velocity along each coordinate, as mentioned in section 2.2 — `Discretize(v_des, v_des_dis)` is simply a nested if statement which sets the value of `v_des_dis`. The error is then the difference between the actual velocity along that axis, and the desired velocity (after discretization).

Each controller is a three state automaton, corresponding to the three possibilities for the thrusters. The logic is quite straightforward — if the error exceeds `high_limit` then the thrusters are fired, and when the error is less than `low_limit` the thrusters are switched off. `Controller=''on''` is used to signal to the global velocity controller that this controller is active, this is used by the global velocity controller to prevent rotation and translation from occurring at the same time. Similarly `State = ``on''` is used by the global controller to signal to this controller that it can proceed with its control — this is switched off if for instance this is a rotational controller, and translation is happening currently.

```
Controller =
 (B1, B2, B3, B4, v_dyn, low_limit, high_limit, saturation)
 [v_des, v_des_dis, State, Controller]  {
  new v_error in new St0 in new St1 in new Stn1 in {
    always {
      Discretize(v_des, v_des_dis),
      v_error:=v_des_dis-v_dyn
    },

    St0(),
    always {
      St0 = (){  /* all thrusters off */
        do always Controller="off"
        watching (v_error >= high_limit || v_error <= -high_limit),
        when (v_error >= high_limit || v_error <= -high_limit) do
          if (v_error >= high_limit) then St1() else Stn1()
      },
      St1 = (){  /* forward pair of thrusters on */
        do always {
          if (State = "on") then {
            B1.voltage=28, B2.voltage=28
          },
          Controller="on"
        } watching (v_error <= low_limit),
        when (v_error <= low_limit) do St0()
      },
      Stn1 = (){  /* backward pair of thrusters on */
        do always {
          if (State = "on") then {
            B3.voltage=28, B4.voltage=28
          },
          Controller="on"
        } watching (v_error >= -low_limit),
        when (v_error >= -low_limit) do St0()
      }
    }
  }
}

Controller(U, T1, T2, T3, T4, Dynamics.u, 0.0001, 0.00015, 0.0762),
Controller(V, T5, T6, T7, T8, Dynamics.v, 0.0001, 0.00015, 0.0762),
Controller(W, T9, T10, T11, T12, Dynamics.w, 0.0001, 0.00015, 0.0762),
Controller(P, T10, T11, T9, T12, Dynamics.p, 0.0001, 0.00015, 0.5236),
Controller(Q, T2, T3, T1, T4, Dynamics.q, 0.0001, 0.00015, 0.5236),
Controller(R, T6, T7, T5, T8, Dynamics.r, 0.0001, 0.00015, 0.5236)
```

The global velocity controller is a two-state finite automaton. It switches between rotation mode and translation mode — it stays in translation mode while there is any translation going on, then it switches to rotation mode, and vice versa. Note that in each mode, three controllers are activated concurrently to control rotation or translation along each axis.

Position Control. The six controllers for rotation and translation in the position control mode are slightly more sophisticated — there are two extra states for representing the fact that the AERCam has reached its maximum velocity on that axis, so should not accelerate further, this is not needed in the velocity mode as the discretization takes care of this. The remaining structure is the same so we omit the actual code.

The global controller for the position control mode takes a target point and a desired direction in which to point after reaching the target. It executes a sequence of 9 states.

1. Compute a rotation to get one axis of the body frame parallel to the direction in which to travel. Break up this rotation into three component rotations, one around each axis. Rotate around the z-axis by activating the z-axis rotation controller, till the z-axis orientation is achieved.

2. Rotate around y-axis, using the y-axis rotation controller.

3. Rotate around x-axis, using the x-axis rotation controller..

4. Fire the appropriate set of thrusters to reach maximum velocity towards the target point (the appropriate set was computed in step 1).

5. Keep cruising (no thrusters to be fired) for the expected time of the journey to the target.

6. Fire thrusters to reach the target point exactly. This is necessary because the small residual errors in the rotations would build up. Corrections on all three axes are done concurrently by the translational controllers.

7. Compute the rotation necessary to point in the desired direction. Rotate around the z-axis.

8. Rotate around y-axis.

9. Rotate around x-axis.

The complexity of this controller primarily lies in the calculations needed to compute the rotations and directions in steps 1 and 7. We implemented a vector and matrix algebra package in Hybrid cc for doing linear algebra operations. We also implemented a package to do quaternion algebra—this included functions to convert quaternions to matrices and vice-versa, compute the product and to split a quaternion into 3, as needed for the first and seventh states.

The advantages of a cc based language were apparent for example in state 6 — here we had three controllers working in parallel to control the translation along different axes, and none of them needed to know about the existence of any other. In a conventional language one would have to interleave their actions, leading to a very complicated control logic. We could have done the rotations concurrently also, except that the quaternion calculations become a lot more complicated.

6 The user interface

The user interacts with the simulation through an animation, which displays the simulation in real time, and allows the user to enter commands. The animation was built using World Tool Kit ®, which is a set of C libraries. The interface consists of a model of the shuttle, along with a model of the AERCam (see Figure 4). The main picture shows both of these, and the user can control the point of view using the mouse. The inset picture is the view of the shuttle from the AERCam's point of view. The fuel level is shown on the side.

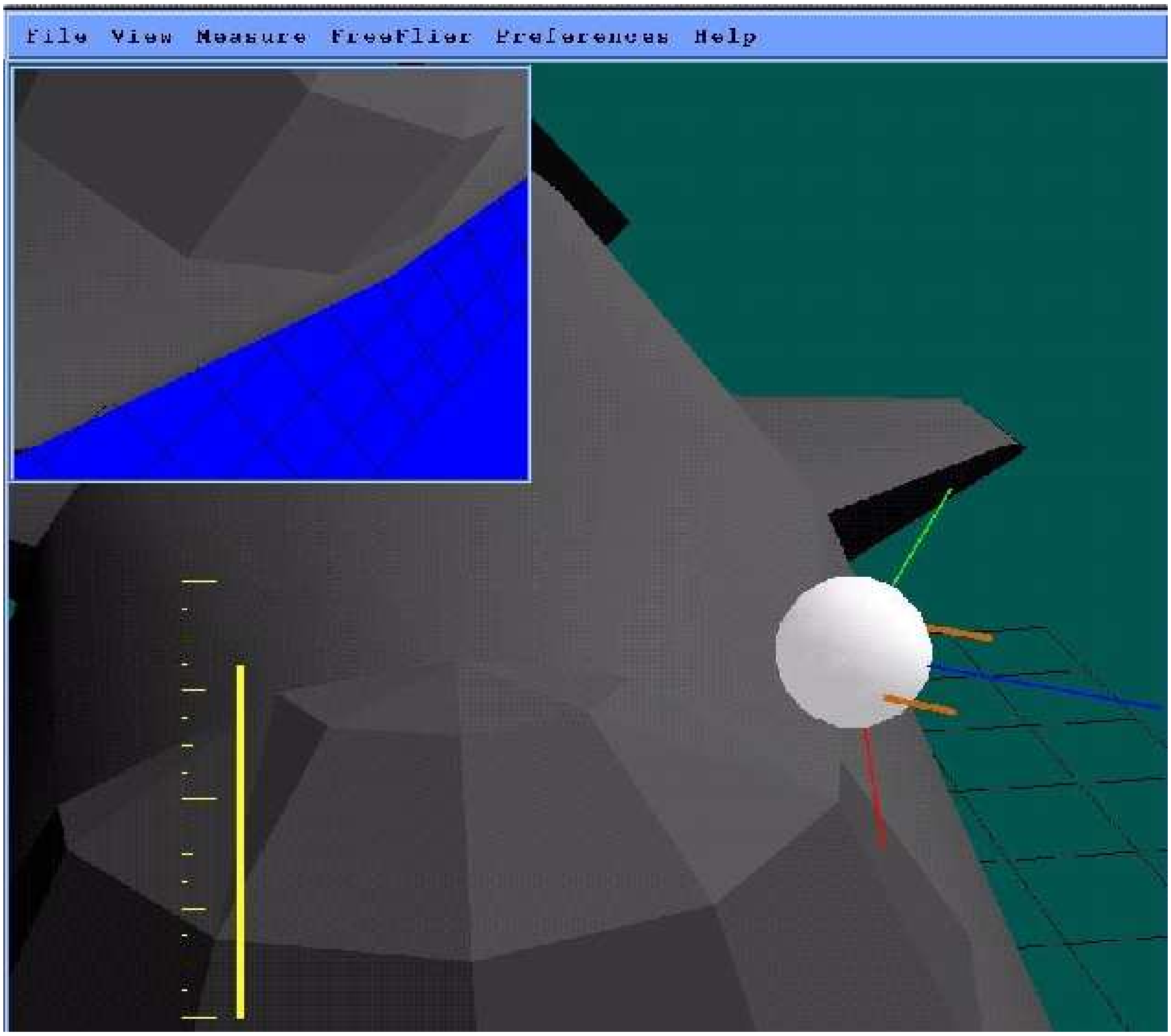


Figure 4: The user interface. The white sphere is the AERCam, the red green and blue lines are the axes in the body frame. The orange lines are the two thrusters that are currently firing. The bar on the left represents the fuel level.

When any thrusters fire, these are shown on the AERCam. The simulation commands available to the user are

- Start simulation. This is always in velocity control mode.
- Suspend simulation. This allows the user to physically move the AERCam to any other location, and make the simulation start from that point.
- Switch to position control mode. In this mode a shadow AERCam is created, which can be moved around by the user to indicate the desired position and orientation. When the user is satisfied by this, he/she can start the simulation, which will take the AERCam to that location.
- Switch to velocity mode. The AERCam can now be controlled by a 3-d mouse, which allows the user to control the translational and angular velocity of the AERCam. The simulation responds with a slight (adjustable) delay. Although the mouse allows both translation and rotation to be specified simultaneously, the simulation allows only one at a time.
- Record the current actions. These are recorded and saved in a file. Later on, this file can be replayed in the viewer like a videotape, at normal speed, fast forward, slow motion and reverse.

The user interface was designed over the Unix pthreads library. The Hybrid cc simulation and the World Tool Kit animation were run as the two separate threads, with information being passed around via shared memory. The information passed from the simulator consists of the position vector, the orientation quaternion, the on/off status of all the thrusters and the fuel level. These are stored in an array, and the animation asks for the values every frame (there were 60 frames per second). An interpolation is done to compute the position values, and this is sent to the animation — we did not reduce the time step of the integration to match the animation as that would involve too many steps and slow down the simulation unnecessarily.

The animation passes any commands from the user to the simulation. In the velocity control mode, this includes a stream of velocity values, which were stored in an array by the simulation, and interpolation was performed whenever Hybrid cc wanted to know the desired velocity at any instant. This is necessary to make the velocity continuous, since the simulator assumes that all arithmetic functions are continuous. In the position control mode, we passed the target position and orientation to the simulator.

Performance. The performance of the system was directly proportional to the number of discrete changes and the time at which they occurred. For instance, in the position control mode, the AERCam takes about 70 seconds to go from $(0, 0, 0)$ to $(-4.2, 0.6, -0.95)$ (including rotations). This involves about 250 point and interval phases, which took a total of 18 seconds on an UltraSparc II. However almost a third of the phases occur during the first 4 seconds of simulation time, as a result during this period the simulation is slightly slower than real time. However as the AERCam starts moving towards its target uniformly (step 5), the simulation was able to catch up, thus overall the simulation was able to keep up with real time easily.

In the velocity mode, the performance of the system depends on the frequency with which the user changes the commands. If the commands are given very frequently (since these are given with a mouse, changing velocity 2-3 times a second is easily possible!), the simulation cannot keep up with the animation. Note that we never slow down the animation, as a result there may be jerks in the position of the AERCam.

Overall we found the performance of the simulation to be quite satisfactory for our purpose. However as models get more complicated, real-time simulation will probably have to be given up in favor of the record and playback simulation.

7 Future work

We plan to use the insights from this modeling effort in building further models for NASA. We have already started building models for a Mars rover — these models will be used for providing a simulation testbed for software, in particular model-based diagnosis and repair software [WN96]. We eventually plan to use the same models for simulation, diagnosis, planning and control.

This effort also highlighted some of the problems in the current version of Hybrid cc, which we will try to overcome. Efficiency is always a problem, and we have made several improvements in the default handling algorithms to increase speed. One important area of research is to try and reduce the amount of work necessary in successive phases by keeping track of the changes that occur, and updating the store rather than recomputing it.

We also learned a lot about programming in Hybrid cc. While programming was not particularly hard — one of us was able to write the basic velocity control after 2 weeks of exposure to Hybrid cc— we did come up with a number of idioms which we will attempt to implement more efficiently in Hybrid cc. An instance of such an idiom is the state machine idiom, which comes up whenever controllers are written.

Acknowledgements. We would like to thanks Charles Neveu for making this project possible, and Ted Blackmon, for his extensive help with the animation.

References

[Arn78] V. I. Arnold. *Mathematical Methods of Classical Mechanics*. Springer Verlag, 1978.

[Ber93] G. Berry. Preemption in concurrent systems. In R. K. Shyamasundar, editor, *Proc. of FSTTCS*, pages 72–93. Springer-Verlag, 1993. LNCS 761.

[Bry94] Arthur E Bryson. *Control of Spaceraft and Aircraft*. Princeton University Press, 1994.

[CG98] Bjorn Carlson and Vineet Gupta. Hybrid CC and interval constraints. In Thomas A Henzinger and Sankar Sastry, editors, *Hybrid Systems 98: Computation and Control*, Lecture notes in computer science, to appear. Springer Verlag, April 1998.

[ER95] Bernard Etkin and Lloyd Duff Reid. *Dynamics of Flight:Stability and Control*. John Wiley and Sons, 1995.

[GJS98] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1-2):3–50, 1998.

[Phi97] Dale Phinney. Aercam sprint flight project. http://tommy.jsc.nasa.gov/projects/Sprint/, 1997.

[Rei80] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81 – 132, 1980.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1993.

[SJG95] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default Timed Concurrent Constraint Programming. In *Proceedings of Twenty Second ACM Symposium on Principles of Programming Languages, San Francisco*, pages 272–285, January 1995.

[WN96] Brian C Williams and P Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*. American Association for Artificial Intelligence, July 1996.