Dependence Analysis and Scheduling for Ordinary Differential Equations

The PREPARE Consortium

Release: 1.1
Date: February 14, 1994
Status: release
Confidentiality: public
Reference: PELAB-2-Scheduling

Contents

1 Introduction

Parallel processing has become a widely used approach for increasing the throughput of modern computers. Development in this area has been promoted by applications that need immense computing power such as weather prediction, chemical modelling and various mechanical simulation problems. Many of these problems are solved by dedicated application programs, which are hard to understand and even harder to port between different parallel architectures without major restructuring or loss in efficiency. The problems should therefore be expressed at the highest possible level of abstraction to reduce the dependence between instances of machines and programs.

The work, which we describe in this paper, is done in ObjectMath [2], a high-level programming environment. It includes symbolic computation and object-oriented constructs which are both good for modelling and very useful when the problem is transformed into a low-level, executable program.

In this report, it is assumed that there are processors, operating systems and imperative languages in which parallelism can be expressed. Instead of having difficulties in finding parallelism in low-level FORTRAN [6, 8, 3] we make an attempt to extract parallelism at a high level where there is more knowledge left about the original problem.

The report is organized as follows. Section 2 gives a short description of ODE-solvers and how they work. Section 3 presents different potential parallelizing techniques. Section 4 describes how the dependence analysis is performed, and in section 5 we use this for scheduling. In section 6 we present three example problem modelled in ObjectMath which we analyse in section 7. The problems in the work are discussed in section 8 and everything is brought to a close in section 9.

1.1 The Problem

A large group of (mathematical) problems can be expressed as initial value problems (IVP) for a system of ordinary differential equations (ODEs) [1]. These problems are solved numerically by applying a good, general, pre-written ODE-solver to the equation system and computing the desired solutions. Existing solvers are inherently sequential due to the methods used. Attempts have been made to restructure and adapt algorithms to extract some parallelism [4, 7].

> **Example:** A ball bounces on a surface. The law of gravity and the forces on the ball when it is in contact with the surface etc. are modelled with ODEs. The state is known when the time is zero and we calculate the position of the ball at different time points. See figure 1.

$$
\begin{array}{ll}
x' = v & x(0) = x_0 \\
v' = \begin{cases} -g & \text{if } x \geq 0 \\ -g - \frac{k_{spring} \cdot x + k_{damp} \cdot v}{m} & \text{if } x < 0 \end{cases} & v(0) = v_0 \\
x(t) = ? &
\end{array}
$$

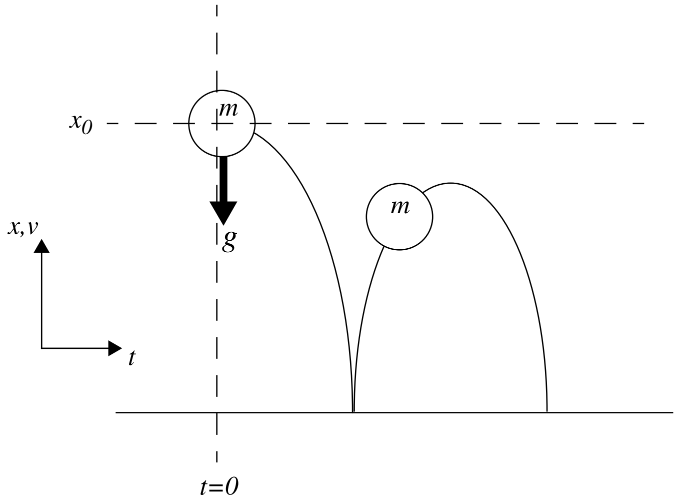

Figure 1: A bouncing ball.

The computed solution of an IVP consists of a large number of calculated approximations where every approximation depends on the previous one. To make the final result exact enough, the accuracy in every calculation must be kept at a high level, forcing the solver to take very small steps and approach the solution at a very slow pace. This makes this kind of computation very time-demanding. The aim here is to shorten the time spent, by using parallelism on different levels.

1.2 Limitations

In this investigation, the possibilities to parallelize the solver itself are omitted since it is a very difficult task to do without weakening the mathematical properties (tolerance, stability, etc.). This is left to the mathematical community. Instead this work is focused on rearranging and partitioning the input parameters and functions required by the solver in order to find parallelism and reduce the time it takes to reach the solutions.

It is assumed that we have a MIMD architecture and an operating system which allows us to express task parallelism.

1.3 Approach

To shorten the computation time we exploit the parallelism in the equations. By using ObjectMath in which the equations can be analysed symbolically, it may be possible to do enough dependence analysis, partitioning and partial evaluation at compile time to efficiently parallelize the calculation. Once we have succeeded in doing this, the remaining problem is to choose an appropriate scheduler which maximize the utilization of the computer. The more that is known about the problem and algorithms the more precise scheduling is possible.

The scheduling algorithms must be fast in order not to decrease the speed too much. We can allow some static scheduling but if dynamic scheduling is needed it must be kept simple since the parallel computation and communication is very intense in this kind of applications (ODE-solvers).

2 ODE-solver

2.1 The Structure

An ODE-solver consists of a general, carefully implemented algorithm which numerically approximates the solution to a given system of ODEs, an initial state, some controlling parameters and solution points where solutions are desired (figure 2).

- The initial state (often trivial to calculate) and control parameters are given once, at the beginning of the computation.
- The desired solution can be either one single point or a hole range of points where solutions are wanted.
- The system of ODEs is a function $\dot{\boldsymbol{y}}(t) = \boldsymbol{f}(\boldsymbol{y}(t), t)$ which calculates the first-order derivatives $\dot{\boldsymbol{y}}(t)$ for each state variable from a given state $\boldsymbol{y}(\mathrm{t})$. Thus the equation system consists of only first-order ODEs. The variable t is called the *free* variable and often represents the time in a simulation. The function should be side-effect free to allow as much parallelism as possible to be extracted.

  From now on we call this function RHS[1].

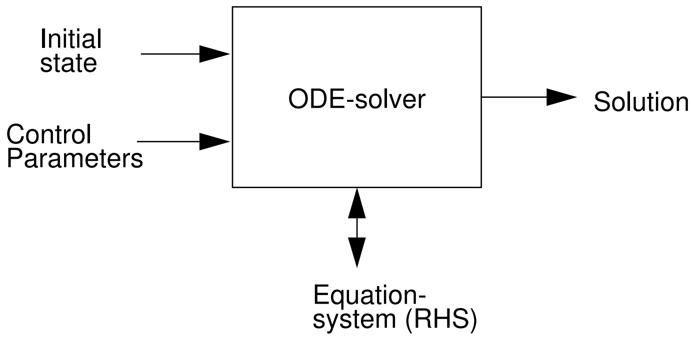

Figure 2: Structure of a ODE-solver for an initial value problem.

2.2 Scheme To Find a Solution

From a known point, $\boldsymbol{y}_n$, on the solution curve, the solver makes an approximation $\boldsymbol{y}_{n+1}$ to $\boldsymbol{y}(t_{n+1})$. This approximation is an extrapolation of either previously calculated points (multistep methods, figure 3a) or intermediate extrapolations (singlestep or Runge-Kutta methods, figure 3b) [7].

[1] Right-Hand-Side

Each time a extrapolation is calculated, the function RHS is invoked. Since the solver usually needs to take a large number of steps to reach the desired solution, the communication between the solver and RHS is intense.

If the method used by the ODE-solver is implicit, the extrapolation point is dependent on itself and calculated by iteration. In that case it can be necessary to calculate the Jacobian matrix, $J = \frac{\partial f(y,t)}{\partial y}$. We can, of course, use the RHS-function to compute an approximation, but it is usually possible to provide the solver with an extra function dedicated to computing the Jacobian.

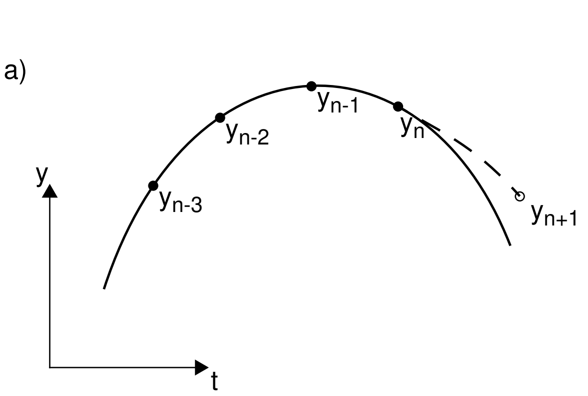

Figure 3: A step in solving an IVP-problem. The left part is a multistep method and the right part is a single-step method

3 Potential Parallelism

There are two ways to exploit parallelism in an IVP-computation:

PAR–1 *Parallelism across the method.*

PAR–2 *Parallelism across the time.*

PAR–3 *Parallelism across the system (or problem).*

The PAR–1 and PAR–2 technique can not be exploited without redesigning the solver. Since this is not in the scope of this report we have left it for future work. This work is aimed at solving the PAR–3 technique. It is based on the given problem and can be classified according to the level of parallelism:

PAR–3.1 *Parallelism at equation system level.* If the set of ODEs can be partitioned into two or more sets which can be solved independently of each other, the computation can be parallelized accordingly.

PAR–3.2 *Parallelism at equation level.* The derivatives in RHS which constitute the equations are independent of each other and can therefore be evaluated in parallel.

When we find two or more sets of ODEs which are independent of each other as the PAR–3.1 technique suggest, we split these sets into separate problems. How independent the sets are is determined by dependence analysis (section 4). The gain of such splitting is:

- If the time to compute the given derivatives is small compared to communication latency, we will get an *increase* in time if we distribute the computation of the derivatives according to the idea of PAR–3.2. With the equation splitting technique (PAR–3.1 we get speedup due to parallelism *even* if the derivatives computation time is short.
- The ODE-solver can, for each ODE system, choose its own step size independently of the others. In each ODE system there are fewer equations and hence fewer bounds on the step size. Consequently, the average step size may increase.
- The ODE-solver's internal computation time decreases due to fewer state variables. If the solver uses an implicit method we can get quadratic speedup thanks to smaller Jacobian matrix.

By investigating the second alternative (PAR–3.2) where RHS is subjected to parallelism we find that this technique makes it easier to utilize the computer. Since RHS is side-effect free (section 2.1) it is possible to calculate the separate derivatives in parallel. This is not in the scope of the report, though, but will be part of the future work.

4 Dependence Analysis

As we mentioned earlier, it is essential to know whether one ODE is dependent on another in order to make the correct partitioning. Here is the definition of dependence:

Definition 1 (variant 1) *An ordinary differential equation* $\dot{y}_1 = f_1(V_1, t)$ *is dependent on another ordinary differential equation* $\dot{y}_2 = f_2(V_2, t)$ iff $y_2 \in V_1$. V_1 *and* V_2 *are sets of state variables.*

Or as a mathematician would express it:

Definition 1 (variant 2) *An ordinary differential equation* $\dot{y}_1 = f_1(V_1, t)$ *is dependent on another ordinary differential equation* $\dot{y}_2 = f_2(V_2, t)$ iff $\frac{\partial f_1}{\partial y_2} \not\equiv 0$.

4.1 Strongly Connected Components

We represent the equation system as a directed graph where the equations are vertices and an arc represents a dependence from one ODE to another. From this graph we extract strongly connected components (SCCs). A strongly connected component is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set. The arcs that go from a vertex in one SCC to a vertex in another SCC are called *cross-component* arcs and the graph where the vertices consist of the SCCs and the only arcs are cross-component arcs is called the *reduced* dependence graph.

Each SCC becomes a system of ODEs which can be given as a separate task to the ODE-solver. The cross-component arcs will result in data transfer betwee the tasks. The order in which the separate systems of ODEs can be solved are partially determined by the cross-component arcs. It is the scheduler's task to finally decide how the computation will proceed.

5 Scheduling

Given the reduced dependence graph, the scheduler finds an order in which the computation will be done. If the architecture is of type MIMD and we have task-parallel constructs in the language, it is trivial: the reduced dependence graph is sorted in topological order and each computation associated with a vertex is started whenever there is a processor free. For every arc that is going into the vertex we do a `Wait` before we start the computation to make sure that the input data is available. In the same way, for every outgoing arc, we do a `Signal` when we are done (figure 4). If there are no possibilities to express task-parallelism, the scheduling problem becomes more difficult since, in most cases, it is very hard to predict how much time it will take to compute a solution to a system of ODEs.

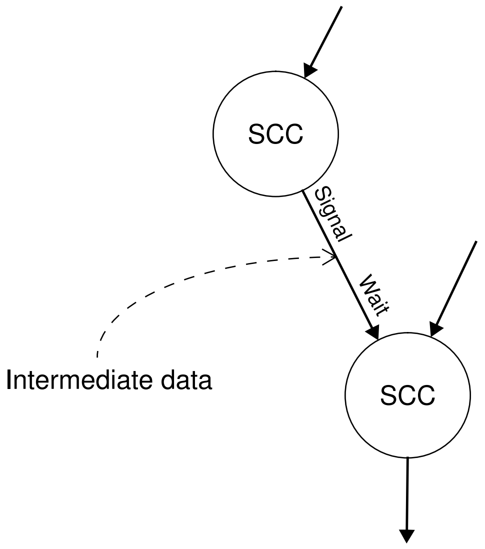

Figure 4: Intermediate dataflow between two strongly connected components.

If the intermediate results between the solution of two systems of ODEs become very large or is not needed by later analysis, we have the opportunity of exploiting pipeline parallelism to accomplish parallel computation with a smaller granularity. In this case we maintain a buffer where we keep the generated, intermediate results which still has not been used by all succeeding ODE-solvers. The synchronization is done at a lower level[2] and we can also limit the size of the buffer in order to save memory space if the amount of data grows.

[2] The level can actually be arbitrary small. It is based on the resolution of the free variable and varies according to the step sizes in the both solvers.

6 Example Applications

In this section, three ObjectMath applications will be presented, with varying degrees of complexity and realism. They have been created as test cases for the ideas previously presented.

When referring to the model's classes, instances and methods, we will adhere to the following convention. Names of classes, instances and methods will be shown like `ClassName`, InstanceName and `MethodName`. Notation specific to a single model will be described in its subsection.

For each model, an object hierarchy will be presented in graphical form. Here, we follow standard ObjectMath conventions: classes are represented by rounded boxes, instances by square boxes, and object parts as boxes within boxes. Inheritance is shown as an arrow pointing from child to parent. Inheritance relations for parts are shown in grey.

6.1 The Four Motor Servo Example

This is a very simple control theory example. The model illustrates how different types of servo respond when an error is introduced.

6.1.1 The Model

There are four servos, each servo consisting of a motor and one or two regulators. The four servos are connected as shown in figure 5. G represents a working motor and G_{err} represents a motor with a problem.

F_1 ... F_4 are different regulators. F_1 and F_3 are chosen in a way that makes the servo with two F_1 regulators identical to the servo with one F_3 regulator. However, when an error is introduced the servos will give different output.

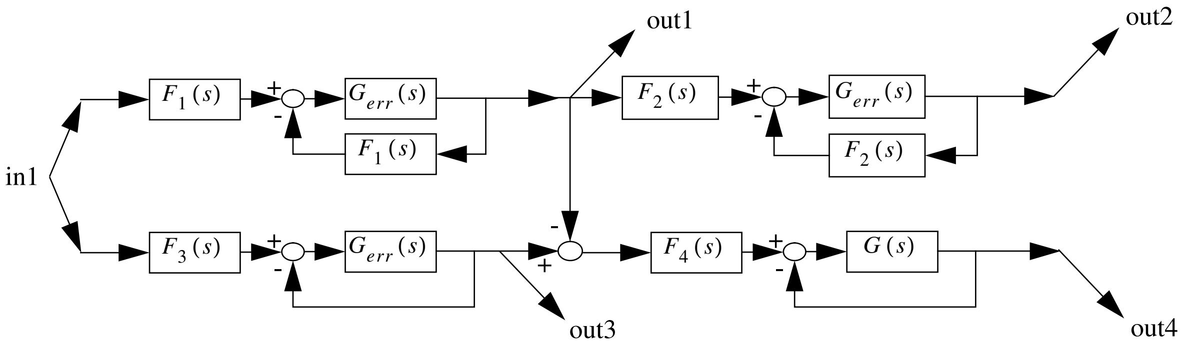

Figure 5: The four motor servo model.

6.1.2 Mathematical Description

Each element, regulator or motor, in the model can be described either by the transition function in the Laplace plane or, as is done in this example, by differential equations describing the relation between input and output.

$G(s)$ and $G_{err}(s)$ are Laplace transforms of the transition function. They are defined as follows, with the corresponding differential equations

$G(s) = \frac{1}{s(s+1)}$ $G_{err}(s) = \frac{1}{s(s+0.5)}$

corresponding to corresponding to

$$\begin{cases} \dot{y}_1 &= u \\ \dot{y}_2 &= -y_2 + u \\ y &= y_1 - y_2 \end{cases} \qquad \begin{cases} \dot{y}_1 &= 2u \\ \dot{y}_2 &= -0.5y_2 + 2u \\ y &= y_1 - y_2 \end{cases}$$

The regulators $F_1(s)...F_4(s)$ are similarly defined as

$$F_1(s) = F_2(s) = 29.7\frac{s+1}{s+9}$$

$$\begin{cases} \dot{u}_2 &= 8x - 9u_2 \\ y &= x - u_2 \end{cases}$$

$$F_3(s) = F_4(s) = 29.7\left(1 - \frac{8s+28.7}{s^2+9s+29.7}\right)$$

$$\begin{cases} \dot{u}_1 &= -9y_1 \\ \dot{u}_2 &= u_1 \\ u &= 29.7\,(-8u_1 - 27.7u_2 + x) \end{cases}$$

6.1.3 The ObjectMath Model

The class hierarchy of the servo model is shown in figure 6. The `motor` class is a general class for motors. `goodmotor` and `badmotor` are classes for motors, defined by the differential equations corresponding to G and G_{err}, respectively.

Similarly, `reg` is a general regulator class and `reg1` and `reg2` are regulators given by the differential equations corresponding to F_1 and F_3, respectively.

`servo1` describes a servo with one motor and two regulators as parts, and `servo2` is a class with one motor and one regulator as parts. The differential equations for an entire servo are generated by combining the differential equations of the parts.

The connections between different servos are defined at the global level. The `savediffs` function in solving is called to save all equations in a format suitable for dependence analysis or solving.

6.2 The Hydroelectric Power Plant

Our second example is also from the domain of control theory. An ObjectMath model of a hydroelectric power plant has been created, including objects like turbines, spillways, dams, and regulators. The model is based on an actual Swedish power plant, *Älvkarleby Kraftverk*, although the faithfulness to the original varies considerably within the model. The focus is

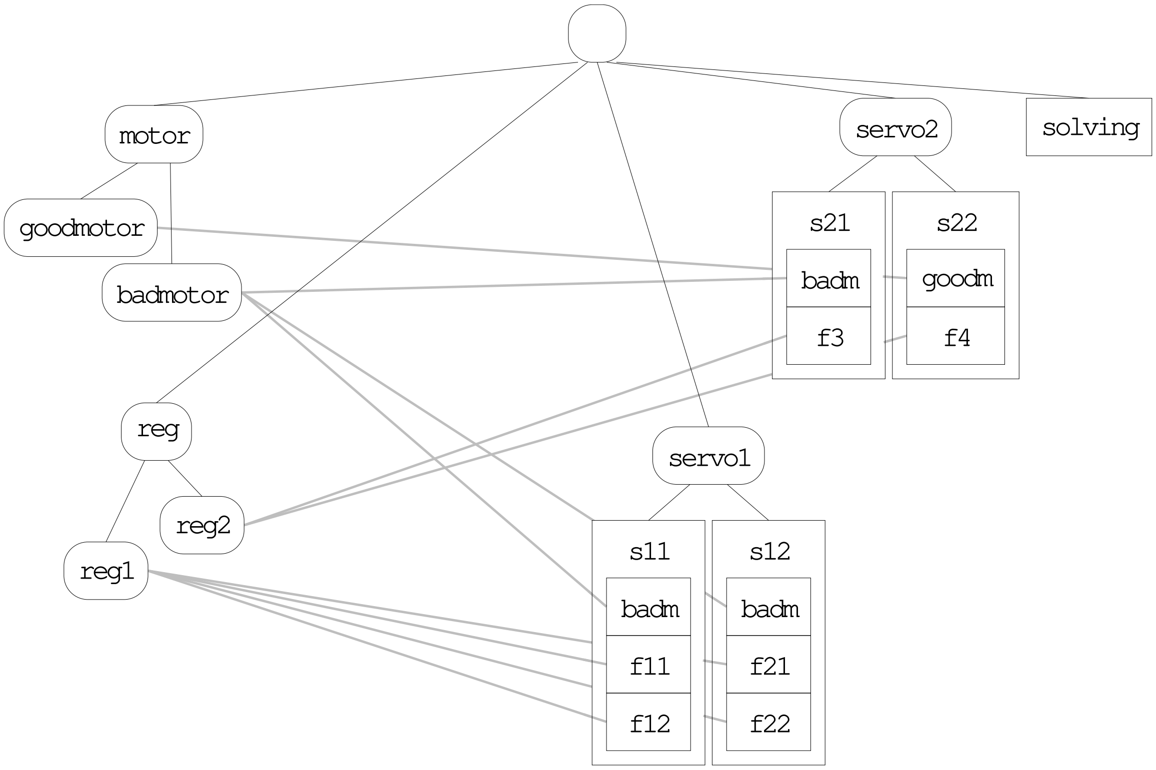

Figure 6: ObjectMath class hierarchy.

on water levels and water flow through the plant. Thus the model could be used for verifying dam safety margins or the effects of surface regulation on the neighbouring salmon farm, for example.

6.2.1 General Information

An ObjectMath specification of the model is included in Appendix B.1. In this description, we will only describe the general structure of the model, and some of the central equations. We will use the same names of classes and instances as in the ObjectMath model. However, for clarity, equations and their variables will be presented in standard mathematical notation.

With a few exceptions, all model variables are time-dependent, so this will not be explicitly stated (although the exceptions will).

6.2.2 Class Structure

Figure 7 shows how the objects that make up the power plant are inherited from object classes, and in some cases aggregated into larger objects. Unique objects (for their class) have been placed at the top of graph. Objects which are part of the water way are placed at the right end of the graph, with objects further down in the water way further to the right.

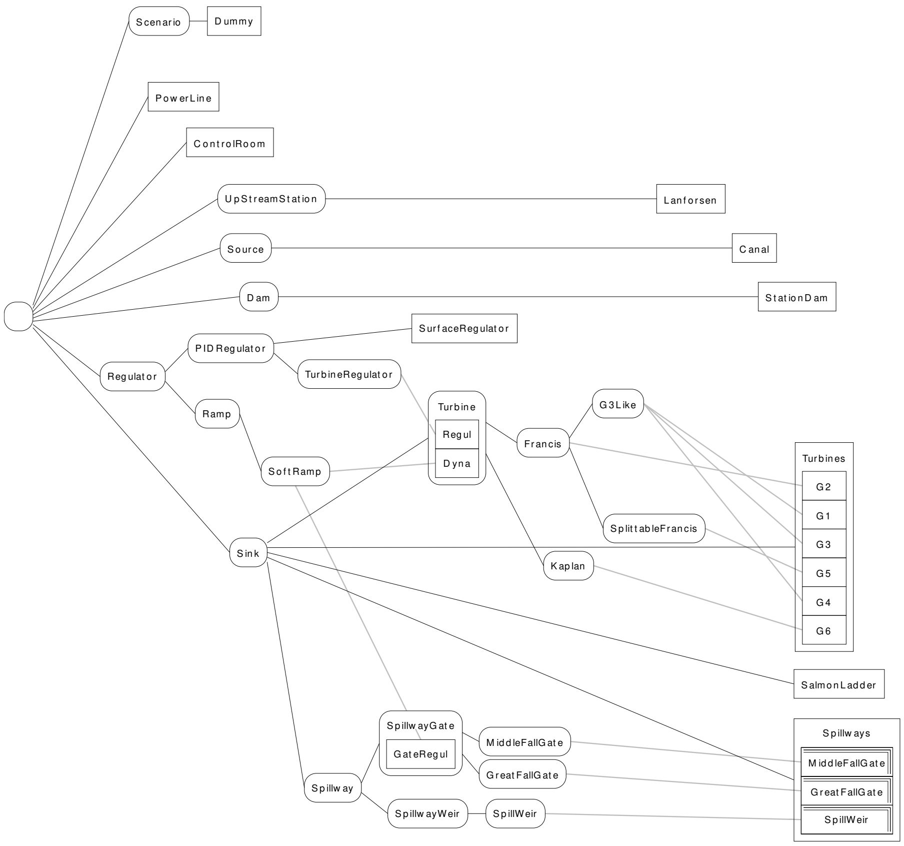

Figure 7: Class hierarchy of the hydroelectric power plant.

The Station Dam StationDam is probably the most central object in the model, connecting most of the other objects to each other. This dam is located above the power station, making it possible to vary the flow of water through the plant within limits set by the riparian court. This particular installation has a very small dam, which means that the limits are tight and surface regulation critical.

The StationDam object is inherited from `Dam`, a class which is characterized by its water surface level, h_{Dam}, and connections to a number of `Drain`s and `Source`s. As expected, these specify the water flow going into the `Dam`, $Q_{i_{Dam}}$, and the outgoing flow, $Q_{o_{Dam}}$.

The geometry of the dam is currently not modelled at all: the surface level is taken to be the same over the entire dam area, with local deviations near `Drain`s. This makes the model unreliable for cases with an extreme, sudden change in the water flow, e.g. when an emergency shutdown is done during the spring flood. The simplification is necessary to avoid the partial differential equations that would result from a better model of the fluid dynamics involved (e.g. using Navier-Stokes' equations).[3]

Instead, the relation between the flow into the dam and the surface level is defined through

[3] Partial differential equations are beyond the scope of this article. Furthermore, they are not supported by the current version of ObjectMath.

a constant, ϑ_{Dam} (called the dam's "hourly value" in hydro power terminology), describing the net input flow required for increasing the surface level one centimetre in one hour.

Water waves in the dam are simulated by adding a simple time-dependent sinewave (with frequency f_w and amplitude A_w) to the surface level.

The resulting differential equation for the dam surface level becomes:

$$h'_{Dam} = \frac{Q_{i_{Dam}} - Q_{o_{Dam}}}{3600 \cdot 100\vartheta_{Dam}} + \frac{d}{dt} A_w \sin 2\pi f_w t$$

Sources The StationDam has only one `Source`. The Canal leading down to it. The other end of the canal is connected to a stream coming from the station upstreams, Lanforsen. The flow coming in through the canal is the $Q_{i_{Dam}}$ mentioned earlier. It is related to the outgoing flow from Lanforsen by delaying it with $T_{Lanforsen}$ seconds:

$$Q_{o_{Canal}}(t) = Q_{o_{Lanforsen}}(t - T_{Lanforsen})$$

Sinks The water flowing out from the StationDam is either making money, going through the `Turbine`s, or wasted going through the `Spillway`s. The latter is of course avoided if possible, but may be necessary when the incoming water flow is greater than the capacity of the available turbines.

Thus, we get a definition of the $Q_{o_{Dam}}$ mentioned earlier:

$$Q_{o_{Dam}} = \sum_{t \in Turbines} Q_t + \sum_{s \in Spillways} Q_s$$

Turbines The power station has six turbines, named G1 through G6. There is one Kaplan turbine whereas the others are Francis turbines. One of the Francis turbines can be splitted which cuts its capacity in half. (This can be valuable at extremely dry periods, when the incoming flow is less than the minimum flow recommended for a full turbine.)

The interesting property of these is how much water they consume at different points in time. This is a function of the throttle setting, $S_{Turbine}$, which in turn is controlled by a turbine regulator included as part of the turbine (in this model). This regulator will be described later in a separate section. The turbine also has some inertia, which is modelled by a part called Dyna.

The relation between throttle setting and water flow is approximated by a second degree polynom, where the constants are specific to the turbine. For e.g. G6 we have:

$$Q_{G6} = \alpha_{G6} S_{Turbine} + \beta_{G6} S_{G6}^2$$

Spillways There are mainly two different kinds of `Spillway`, called `SpillwayGate` and `Spillweir`. The latter type is only intended for extreme situations: when the surface reaches a certain level, water will simply splash over the spillweir threshold. On the other hand, `SpillwayGate`s have motorized sector gates, which can be opened as required, down to a $\frac{1}{20}$ of a degree. The motor in each gate is controlled by a part called `GateRegul`, which is a simple on/off regulator.

In both cases, the resulting water speed, $v_{w_{Gate}}$, depends on the surface level, as stated in Bernoulli's law:

$$v_{w_{Gate}} = \sqrt{2g\frac{h_{Dam} - h_{Threshold_{Gate}}}{2}}$$

The resulting flow depends on the spill area, which is a function of the gate/weir geometry. For details, see appendix B.1.

The Salmon Ladder Some water flow is necessary for the salmon ladder. This is controlled by an automated gate, which is set so that a constant flow of water is maintained at all times. We model this by having a simple parameter describing this flow.

Regulators There are two kinds of regulator, the `Ramp` regulator and the `PIDRegulator`. These are connected to

- A reference input, $r(t)$, indicating what some variable in the system ought to be.
- The actual input, $y(t)$, indicating what it really is. The error is defined as $e(t) = r(t) - y(t)$.
- The output, $u(t)$, which is how the regulator influences the system to reach the target reference value.

The `Ramp` regulator is an extremely simple regulator, which raises/lowers its output value at a constant rate, until the reference value matches the actual value. It is the parent of the `SoftRamp` which works in a similar manner, with the difference being that the `SoftRamp` adds inertia to the system, slowing down the transition between *constant output* and *constant increase/decrease* rate. `SoftRamp` is used to model simple inertial systems (gates and throttle valves), which are controlled by an on/off regulator. Admittedly, using it like that is not altogether pure from a modelling point of view, but it is convenient.

The `PIDRegulator` implements a standard PID-regulator. We use the standard definition for an analog, time-continuous regulator, although the actual regulators in our model are implemented digitally. Again, this is necessary to restrict the problem to the domain which can be expressed as a system of ordinary differential equations. However, the errors resulting from this approximation are slight. With the regulator parameters P, T_I, and T_D we have:

$$u(t) = P(e(t) + \frac{1}{T_I}\int_{t_o}^{t} e(\tau)d\tau + T_D\frac{de(t)}{dt})$$

The SurfaceRegulator is an instance inherited from `PIDRegulator`. It can be configured to control any number of turbines, changing their flow in order to achieve a certain surface level. However, we have chosen to keep this configuration fixed for the entire length of a simulation. This is in line with the way surface regulation is normally used: the choice of turbines used for this purpose does not change very often. In the dependence analysis (see section 7) we choose G6 for surface regulation purposes.

Each turbine has a `TurbineRegulator` which controls its throttle valve, trying to reach a certain throttle percentage. However, the regulation is also somewhat influenced by the mains frequency, which should be as close as possible to 50 Hz. To achieve this, all hydro power stations try to compensate frequency fluctuations, in a small way. This is implemented by a compensation term, proportional to the frequency drift, which is added to the regulator's reference value. The relation between frequency drift and the compensation term can be configurable in four discrete settings, EP0 through EP3. EP3, which gives the largest compensation, is used in emergency cases, where the frequency drift is much too large. This setting also adjusts the T_I parameter of the PID-regulator.

The Scenario `Scenario` is a class which enumerates all "free" variables in the model. Objects inheriting from it specify a scenario, where some of these variables are fixed for the duration of the simulation. It is usuallypossible to fix some variables, and doing so may yield substantial performance improvement. The appropriate scenario is chosen by a class method called `select[]`.

6.3 Cylindrical Roller Bearing Model

A simplified rolling bearing model has been created. The SKF Engenering & Research Centre B.V. has implemented a larger model by hand. The geometry of the bearing is shown in Figure 8. It is a two-dimensional (2D) model consisting of an outer ring (Or), an inner ring (Ir) and a single roller (Ro), all defined in a global coordinate system, GCS. The z-axis (not visible) is pointing inwards and the coordinate system is right-oriented. Angles are measured clockwise from the x-axis.

In the current model, the Ir is fixed: its origin coincides with the origin of the GCS. The position and rotation of the Or are inputs to the model. We are mainly interested in describing the dynamic motion of the roller Ro. This motion can be described by the solution to a system of ordinary differential equations (ODEs), as implied by Newton's first and second law.

6.3.1 The Contact Coordinate System

There are two contact cases: the roller against either the inner ring or the outer ring. For each contact case, two contacts are defined: roller against ring, and ring against roller ($(Ro, Ir), (Ro, Or), (Ir, Ro), (Or, Ro)$). For each contact, a local contact coordinate system (CCS) is defined, relative to the GCS. The x-axis of the CCS intersect with the geometrical centre of the two bodies. The origin of the CCS is located at the contact point, where the bodies interact.

The position and orientation of all coordinate systems and bodies are defined by their respective ***origin*** and ***turn***, given relative to the GCS.

6.3.2 The Contact Variables

Only a brief description is given here. For the exact details we refer to the ObjectMath code in appendix C.1.

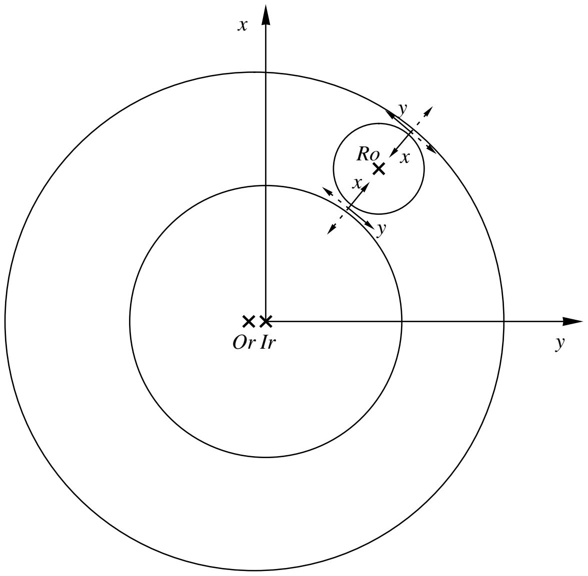

Figure 8: Geometry of the rolling bearing.

For the roller, $\boldsymbol{origin}$ is represented by $(R\cos\theta, R\sin\theta, 0)$ and $\boldsymbol{turn}$ by $(0, 0, \varphi)$ i.e. the position of its body is given in cylindrical coordinates. These quantities are time-variant.

In the following, an index like $_{(Ro,Ir)}$ refers to some property of Ro in the contact between Ro and Ir. In contrast, the index $_{\{Ro,Ir\}}$ refers to something that is *common* to Ro and Ir. It could just as well be written $_{\{Ir,Ro\}}$. These indices are often omitted where the context makes it clear what is intended, or when the index does not matter. Also, coordinate systems and transformations between them are not stated explicitly.

The angle $\boldsymbol{turn}_{(Ro,Ir)[z]}$ of the local coordinate system, i.e. the direction angle of the x-axis, is the same as the direction angle of

$$\boldsymbol{dist}_{(Ro,Ir)} = \boldsymbol{origin}_{Ro} - \boldsymbol{origin}_{Ir}.$$

The $\boldsymbol{origin}_{(Ro,Ir)}$ of the local system is located at the contact point (see figure 9).

The bodies can have geometrical imperfections. Specifically, the radius may be somewhat wavy, which is modelled by sine wave superposition.

$$radius(\nu)_{(Ro,Ir)} = radius_{nom,Ro} + amp_{Ro}\sin(nwaves_{Ro} \cdot \nu_{(Ro,Ir]})$$

$\nu_{(Ro,Ir)}$ is calculated in almost the same way as $\boldsymbol{turn}_{(Ro,Ir)[z]}$. The intersection distance Δ is calculated as:

$$\Delta_{\{Ro.Ir\}} = \|\boldsymbol{dist}_{(Ro,Ir)or(Ir,Ro)}\| - radius_{Ro,Ir} - radius_{Ir,Ro}$$

The contact point is located in the exact middle between the bodies. If Δ is positive, there is neither any force nor any torque. We define a lever, taking the intersection into consideration:

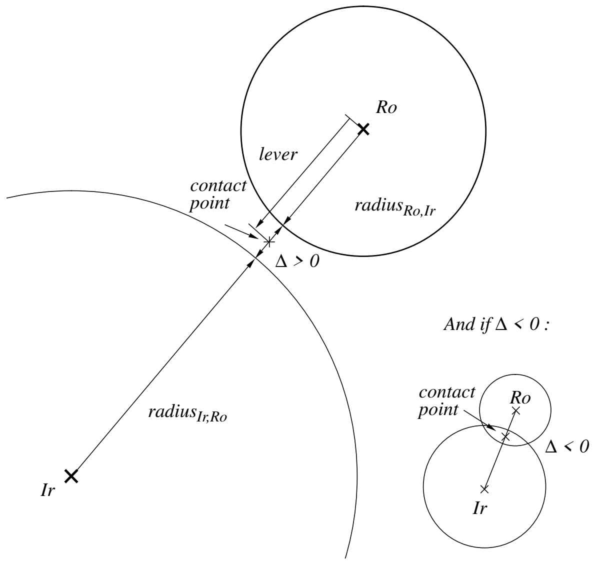

Figure 9: Detail picture of the bearing.

$$lever_{(Ro,Ir)} = radius_{(Ro,Ir)} + \Delta_{\{Ro,Ir\}}/2$$

$\boldsymbol{v}_\Delta$, the relative motion of Ro against Ir at the contact point is calculated as:

$$\begin{aligned}
\boldsymbol{v}_{contact(Ro,Ir)} &= \frac{d}{dt}\boldsymbol{origin}_{Ro} + \frac{d}{dt}\boldsymbol{turn}_{Ro} \times (-lever_{(Ro,Ir)}, 0, 0)^T \\
\boldsymbol{v}_{\Delta(Ro,Ir)} &= \boldsymbol{v}_{contact(Ir,Ro)} - \boldsymbol{v}_{contact(Ro,Ir)}
\end{aligned}$$

$\boldsymbol{v}_\Sigma$, the sum of the relative body motions towards the point of contact (= the ***origin*** of the local coordinate system) is:

$$\begin{aligned}
\boldsymbol{v}_{rel(Ro,Ir)} &= \boldsymbol{v}_{contact(Ro,Ir)} - \frac{d}{dt}\boldsymbol{origin}_{(Ro,Ir)} \\
\boldsymbol{v}_{\Sigma\{Ro,Ir\}} &= \boldsymbol{v}_{rel(Ro,Ir)} + \boldsymbol{v}_{rel(Ir,Ro)}
\end{aligned}$$

6.3.3 The Contact Forces

The following describes how a force $\boldsymbol{F}_{(Ro,Ir)}$ and torque $\boldsymbol{M}_{(Ro,Ir)}$ are calculated, i.e. the force and torque on Ro caused by the contact between Ro and Ir.

The forces and torques are calculated as follows:

$$F_{elastic(Ro,Ir)} = \begin{cases} 0 & ,\Delta \geq 0 \\ \frac{2^{3/2}}{3} \cdot \frac{E}{1/radius_{nom,Ro} + 1/radius_{nom,Ir}} \cdot (-\Delta)^{3/2}, \quad E = 2.26 \cdot 10^{11} & ,\Delta < 0 \end{cases}$$

$$
\begin{aligned}
F_{normal(Ro,Ir)} &= F_{elastic(Ro,Ir)} \cdot (1 - 80\% \cdot smoothsign(\boldsymbol{v}_{\Delta(Ro,Ir)[x]}, 700)) \\
F_{slip(Ro,Ir)} &= -\mu_0 \cdot smoothsign(\boldsymbol{v}_{\Delta(Ro,Ir)[x]}, 8 \cdot 10^3) \cdot F_{normal}
\end{aligned}
$$

$$
smoothsign(a, b) = \tanh ab
$$

($smoothsign$ is a smoothed sign function, having a derivative of b in $a = 0$.)

$$
\begin{aligned}
\boldsymbol{M}_{slip(Ro,Ir)} &= (-lever_{(Ro,Ir)}, 0, 0)^T \times \boldsymbol{F}_{(Ro,Ir)} \\
M_{roll(Ro,Ir)} &= F_{normal(Ro,Ir)} \cdot 2.1 \cdot 10^{-6} \cdot smoothsign(\boldsymbol{v}_{\Sigma(Ro,Ir)[y]}, 700)
\end{aligned}
$$

Several forces and torques are taken into consideration, all given in the local coordinate system.

$$
\begin{aligned}
\boldsymbol{F}_{(Ro,Ir)} &= (F_{normal(Ro,Ir)}, F_{slip(Ro,Ir)}, 0)^T \\
\boldsymbol{M}_{(Ro,Ir)} &= \boldsymbol{M}_{slip(Ro,Ir)} + (0, 0, M_{roll(Ro,Ir)})^T
\end{aligned}
$$

The force and torque acting on the roller is a sum of contact and external forces/torques.

$$
\begin{aligned}
\boldsymbol{F}_{Ro} &= \boldsymbol{F}_{(Ro,Ir)} + \boldsymbol{F}_{(Ro,Or)} + \boldsymbol{F}_{ext} \\
\boldsymbol{M}_{Ro} &= \boldsymbol{M}_{(Ro,Ir)} + \boldsymbol{M}_{(Ro,Or)}
\end{aligned}
$$

$\boldsymbol{F}_{ext}$ represents the gravitational force in this model.

The force F_{Ro} and torque M_{Ro} acting on the roller are calcutated, and are used in a system of differential equations to determine the ***origin*** and ***turn*** of Ro.

$$
\left\{
\begin{array}{lllll}
m\ddot{\boldsymbol{x}} = \boldsymbol{F}, & \boldsymbol{x} = (R\cos\theta, R\sin\theta, 0)^T, & R > 0 \\
I\ddot{\varphi} = M, & M = \boldsymbol{M}_{[z]}
\end{array}
\right.
$$

leading to

$$
\left\{
\begin{aligned}
\ddot{R} &= (\cos\theta, \sin\theta, 0)^T \cdot \frac{\boldsymbol{F}}{m} + R\dot{\theta}^2 \\
\ddot{\theta} &= ((\cos\theta, -\sin\theta, 0)^T \cdot \frac{\boldsymbol{F}}{m} - 2\dot{\theta}\dot{R})/R \\
\ddot{\varphi} &= M/I
\end{aligned}
\right.
$$

6.3.4 The Objectmath Model

Figure 10 shows the class hierarchy of the bearing model. The `Bearing` class embodies the whole model, inheriting from all other classes. The instance `B` is the most complete bearing model. The other bearing variants are simpler special cases. `B2` has no “noise” on the body surfaces. `B1` simplifies things even more by having Or's and Ir's ***origins*** coincide. Ro's motion is only affected by the gravitational force.

`SpinningElement` is an abstract class that contains only the names ***origin*** and ***turn***.

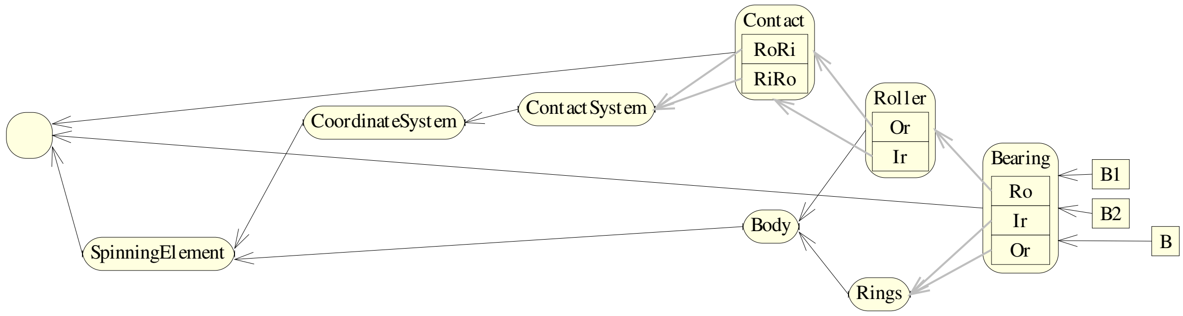

Figure 10: ObjectMath class hierarchy.

`CoordinateSystem` defines a transformation matrix from a local coordinate system to the global one, in terms of $\boldsymbol{turn}_{[z]}$. This matrix is used in a method that transforms a vector from the local to the global system and vice versa.

`ContactSystem` defines the $\boldsymbol{turn}$ and $\boldsymbol{origin}$ for a local coordinate system. Together with `Contact` it defines things that are common to the two coordinate systems used in the contact. Thus, it defines the force and torque in that system, and in principle all the mathematics as described above.

`Body` defines a the location and orientation of a body.

$$\boldsymbol{origin} = (R\cos\theta, R\sin\theta, 0), \quad \boldsymbol{turn} = (0, 0, \varphi)$$

R, θ and φ are the unknown time-dependent functions in the differential equation for Ro. For Ir and Or, they are given explicitly as input.

They also define the "noisy" radius, mass, intertia, gravitation force, and the system of differential equations (used later for Ro only). To calculate the initial states to the differential equations, we initially set

$$\begin{cases} \dot{R} &= 0 \\ \theta &= 0 \\ \varphi &= 0 \end{cases}$$

and thereafter numerically solve the equations of equlibrium

$$\begin{cases} \ddot{R} &= (\cos\theta, \sin\theta, 0)^T \cdot \frac{\boldsymbol{F}}{m} + R\dot{\theta}^2 &= 0 \\ \ddot{\theta} &= ((\cos\theta, -\sin\theta, 0)^T \cdot \frac{\boldsymbol{F}}{m} - 2\dot{\theta}\dot{R})/R &= 0 \\ \ddot{\varphi} &= M/I &= 0 \end{cases}$$

at $t =' initial\ time'$, to obtain R, $\dot{\theta}$ and $\dot{\varphi}$.

`Rings` define

$$\begin{array}{rcl} \boldsymbol{F}_{this} &=& \boldsymbol{F}_{(this,Ro)} + \boldsymbol{F}_{ext} \\ \boldsymbol{M}_{this} &=& \boldsymbol{M}_{(this,Ro)} \end{array} \quad \text{where } this \in \{Ir, Or\}.$$

However, these definitions are not used in the current implementation. The differential equations for the rings ar not used.

`Roller` contains two contacts, one with Ir and one with Or. The forces and toques from their `ContactSystem` parts, that have Ro as its "first" body, are added together:

$$\begin{aligned} \boldsymbol{F}_{Ro} &= \boldsymbol{F}_{(Ro,Ir)} + \boldsymbol{F}_{(Ro,Or)} + \boldsymbol{F}_{ext} \\ \boldsymbol{M}_{Ro} &= \boldsymbol{M}_{(Ro,Ir)} + \boldsymbol{M}_{(Ro,Or)} \end{aligned}$$

Also, it contains redefinitions of the radius (without "noise") and mass, taken from the defaults defined in `Body`.

`Bearing` contains default values for $R(t)$, $\theta(t)$, $\varphi(t)$, $nwaves$ and amp. The latter two model the surface wavyness of Ir and Or.

This class also contains the method `OutFile`, which outputs the differential equation in the form required for the numerical solver and the dependence analyzer. For the complete bearing `B`, this equation amounts to $\sim$300 kB of data.

Also included is a method that solves the differential equations in Mathematica and plots the result. However, it is only useful for the simplest bearing variant `B1`, because of Mathematica's immense space and time requirements.

7 Analysis of the Examples

From the three ObjectMath models above we extract the expressions needed to do a simulation. We feed them to a dependence analyser we have built to obtain the SCCs. The *EDGE* graph editor [5] is used to visualize the dependence graph.

7.1 The Servo

This example is somewhat artificial. The formulae for the regulators and motors are in accordance with reality to a certain degree, but the way the parts are connected together is in this example entirely fictitious. However, it is a good example for showing parallelism at equation system level.

In figure 11 we see that there are eight rather small SCCs. We also see their mutual dependences, the direction of them and which SCCs that are involved. From this it is easy to determine which variables that must be kept as intermediate results. There is a rather high degree of parallelism at equation system level in this example.

7.2 The Hydroelectric Power Plant

As mentioned earlier, this example is based on a real power plant. The focus of interest is directed towards the single turbine which is connected to the surface regulator, in this example G6 (figure 12). The other five turbines are supposed to run at their maximum efficiency level and only respond to changes in the net frequency. All the spillways are modelled by one trivial ODE each. These are typically solved symbolically beforehand and not simulated by the ODE-solver. Since the dependence graph is very wide there is a opportunity of achieving parallelism.

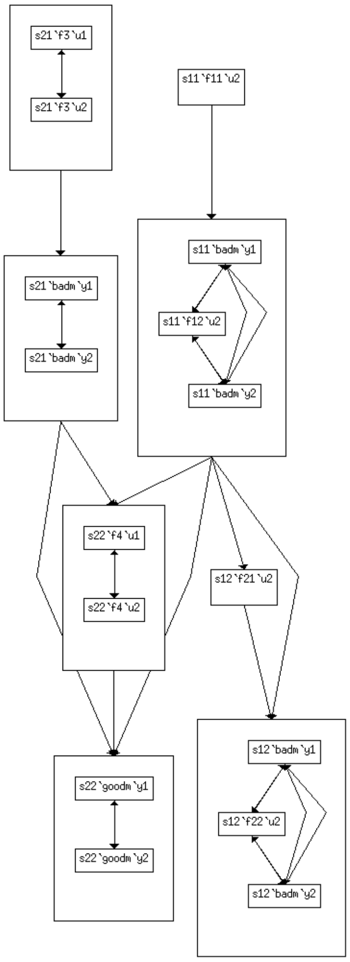

Figure 11: The dependencies of the ODEs in the servo example.

7.3 The Rolling Bearing

The Rolling Bearing is the most complex example from a mathematical point of view. All ODEs are strongly connected except one. The only equation that no other is dependent upon describes the angular position of the rolling element in its own coordinate system. Since it is absolutely round, its rotation angle has no relevance for the rest of the system. Hence it is probably not interesting to calculate, except if the user is interested in viewing it.

8 Discussion

Is the dependence analysis we have done useful in our parallelizing process? If we assume that it is possible to gain speed by partitioning the ODE-system, then the dependence analysis is always useful. Thus the question we really should ask is whether there are real problems

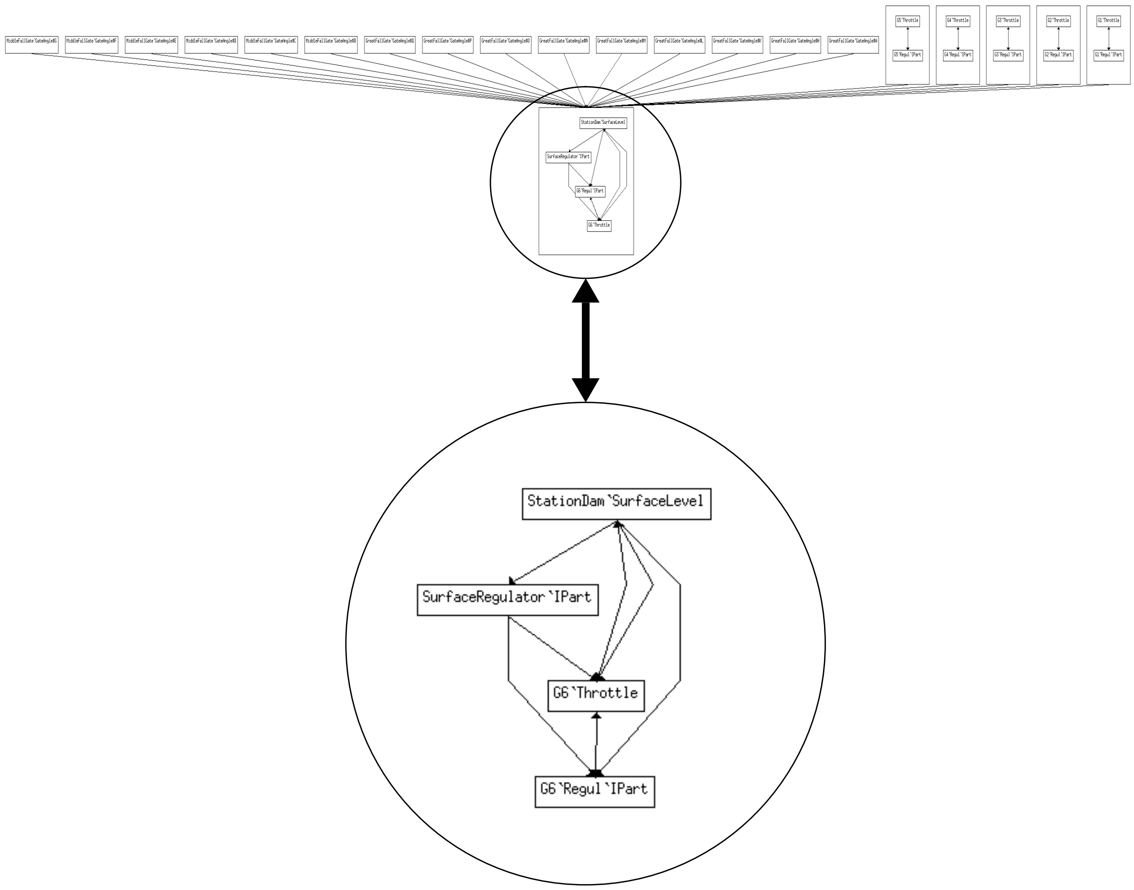

Figure 12: The dependences of the ODEs in the hydro plant example.

that can be partitioned. Looking at the examples[4] we see that there really are more than one SCC. A closer examination of the SCCs shows that there are usually one SCC where the "main" problem is located, and one or more peripheral SCCs which are either trivial or uninteresting to calculate (or both). To use this analysis as the only guide for parallelization seems to be futile since using it on a real problem doesnot yield much parallelism.

However, the analysis and the visualisation are very helpful tools for the problem implementor. It is easy to find missing dependences or dependences that should not be there. Also, uninteresting parts of the problem can be removed at an early stage so that no computing power is wasted.

Though the dependence analysis appears to be of little use, the analysis *can* be useful in collaboration with other analyses:

- From definition 1 in section 4 we have that a dependence is the same as if the partial derivative is equivalent with zero. If we use an implicit solver and need to calculate the Jacobian there could be elements in the matrix which are always zero and not necessary

[4] Ignoring the first one since that is not based on a real problem.

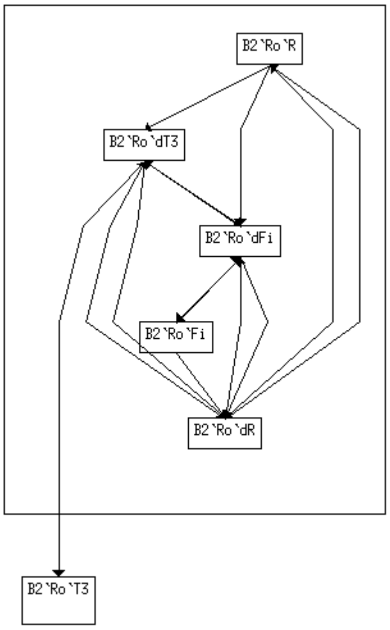

Figure 13: The dependencies of the ODEs in the rolling bearing example.

to calculate. This information can easily be deduced from the dependence graph.

- The cross-component arcs in the reduced graph show where communication is needed. When using the PAR–3.2 parallelizing technique to distribute RHS, the dependence analysis can guide the initialization of the communication links.

- The dependences can be dependent on the free variable (here: the time). By partitioning the solution in time and looking at each part separated from the others, several dependences may disappear. By detecting non-continuous functions in the implementation of RHS, a suitable partitioning can be selected. Recall the bouncing ball example in section 1.1: when the ball is not in contact with the surface, it is not dependent on the surface and therefore some if its computation can be ignored.

9 Conclusions and Future Work

We have described a technique that we supposed would be useful for extracting parallelism from an initial value problem of ordinary differential equations. This assumption was found to be impractical as the only technique to rest upon. This is due to the often high level of connectivity in *real* problems. However, as an intermediate analysis in other parallelizing techniques it appears to be useful.

The three examples in section 6 has been modelled in ObjectMath, a programming environment which includes symbolic computation. By using a system where symbolic simplification is available we get a spin-off effect: we do not need to simplify the expressions in the

ODEs ourself which saves us a lot of dull work.[5]

We will continue the work trying to achieve higher degree of parallelism by parallelize at equation level. It seems to be more applicable to real problems compared to parallelizing at equation system level. The positive cooperative effects of the ideas in section 8 will be further investigated. We will also look at parallelism in other problem areas and try to generalize the parallelizing problem.

References

[1] Francois E. Cellier. *Continuous System Modeling.* Springer-Verlag, 1991.

[2] Peter Fritzson, Vadim Engelson, and Lars Viklund. Variant handling, inheritance and composition in the ObjectMath computer algebra environment. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 145–160. Springer-Verlag, 1993.

[3] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transaction on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[4] Yi ling F. Chiang, Ji-Suing Ma, Kuo-Lin Hu, and Chia-Yo Chang. Parallel multischeme computation. *Journal of Scientific Computing*, 3(3):289–306, 1988.

[5] Frances Newbery Paulisch and Walter F. Tichy. EDGE: An extendible graph editor. *Software—Practice and Experience*, 20(S1):63–88, June 1990.

[6] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transaction on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[7] P. J. van der Houwen and B. P. Sommeijer. Iterated Runge-Kutta methods on parallel computers. *SIAM J. Sci. Stat. Comput.*, 12(5):1000–1028, September 1991.

[8] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transaction on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

A Servo example

A.1 servo.omt (ObjectMath code)

```
MODEL servo;
  s21'in := in1[t];
  s11'in := in1[t];
```

[5] But we do have to proceed cautiously: in the bearing model, which is a very small one compared to *real* problems, we ran into problems because of growth of the expressions in the simplification process.

```
  s12`in := s11`out;
  s22`in := s21`out - s11`out;

  out1 := s11`out;
  out2 := s12`out;
  out3 := s21`out;
  out4 := s22`out;

CLASS motor
END motor;

CLASS goodmotor INHERITS motor
  diffs := { y1' == u, y2' == -y2+u };
  y := y1-y2;
  u := in;
  out := y;
END goodmotor;

CLASS badmotor INHERITS motor
  diffs := { y1' == 2u, y2' == -0.5y2+2u };
  y := y1-y2;
  u := in;
  out := y;
END badmotor;

CLASS reg
END reg;

CLASS reg1 INHERITS reg
  diffs := { u2' == 8x-9u2 };
  u := x-u2;
  x := in;
  out := u;
END reg1;

CLASS reg2 INHERITS reg
  diffs := { u1' == -9u1-29.7u2+x, u2' == u1 };
  u := 29.7(-8u1-28.7u2+x);
  x := in;
  out := u;
END reg2;

CLASS servo1(m,r1,r2)
  out := m`out;
  m`in := r1`out - r2`out;
  r2`in := m`out;
  r1`in := in;
  diffs := Join[m`diffs, r1`diffs, r2`diffs];
END servo1;
```

```
CLASS servo2(m,r)
  out := m`out;
  m`in := r`out - m`out;
  r`in := in;
  diffs := Join[m`diffs, r`diffs];
END servo2;

INSTANCE s11 INHERITS servo1(`badm,`f11,`f12)
PART badm INHERITS badmotor;
PART f11 INHERITS reg1;
PART f12 INHERITS reg1;
END s11;

INSTANCE s12 INHERITS servo1(`badm,`f21,`f22)
PART badm INHERITS badmotor;
PART f21 INHERITS reg1;
PART f22 INHERITS reg1;
END s12;

INSTANCE s21 INHERITS servo2(`badm,`f3)
PART badm INHERITS badmotor;
PART f3 INHERITS reg2;
END s21;

INSTANCE s22 INHERITS servo2(`goodm,`f4)
PART goodm INHERITS goodmotor;
PART f4 INHERITS reg2;
END s22;

INSTANCE solving
  alldiffs := Join[s11`diffs, s12`diffs, s21`diffs, s22`diffs];

  savediffs := Module[ { outfile = OpenWrite["diffs.math"] },
 WriteString[outfile,
     ToString[FullForm[Simplify[alldiffs]]]];
 Close[outfile] ];
END solving;
```

A.2 servo.edge (dependency data)

```
graph: {
  node.borderwidth: 1
  edge.thickness: 1
  /* list of SCC */
  graph: { title: "s22`goodm`y1\ns22`goodm`y2\n"
    /* list of nodes */
    node: { title: "s22`goodm`y1" }
    node: { title: "s22`goodm`y2" }
```

```
    /* list of edges */
    edge: { sourcename: "s22`goodm`y1"  targetname: "s22`goodm`y2" }
    edge: { sourcename: "s22`goodm`y2"  targetname: "s22`goodm`y1" }
  }
  graph: { title: "s22`f4`u2\ns22`f4`u1\n"
    /* list of nodes */
    node: { title: "s22`f4`u2" }
    node: { title: "s22`f4`u1" }
    /* list of edges */
    edge: { sourcename: "s22`f4`u1"  targetname: "s22`f4`u2" }
    edge: { sourcename: "s22`f4`u2"  targetname: "s22`f4`u1" }
  }
  graph: { title: "s21`badm`y1\ns21`badm`y2\n"
    /* list of nodes */
    node: { title: "s21`badm`y1" }
    node: { title: "s21`badm`y2" }
    /* list of edges */
    edge: { sourcename: "s21`badm`y1"  targetname: "s21`badm`y2" }
    edge: { sourcename: "s21`badm`y2"  targetname: "s21`badm`y1" }
  }
  graph: { title: "s21`f3`u2\ns21`f3`u1\n"
    /* list of nodes */
    node: { title: "s21`f3`u2" }
    node: { title: "s21`f3`u1" }
    /* list of edges */
    edge: { sourcename: "s21`f3`u1"  targetname: "s21`f3`u2" }
    edge: { sourcename: "s21`f3`u2"  targetname: "s21`f3`u1" }
  }
  graph: { title: "s12`f22`u2\ns12`badm`y1\ns12`badm`y2\n"
    /* list of nodes */
    node: { title: "s12`f22`u2" }
    node: { title: "s12`badm`y1" }
    node: { title: "s12`badm`y2" }
    /* list of edges */
    edge: { sourcename: "s12`badm`y1"  targetname: "s12`f22`u2" }
    edge: { sourcename: "s12`badm`y2"  targetname: "s12`f22`u2" }
    edge: { sourcename: "s12`badm`y1"  targetname: "s12`badm`y2" }
    edge: { sourcename: "s12`f22`u2"  targetname: "s12`badm`y2" }
    edge: { sourcename: "s12`badm`y2"  targetname: "s12`badm`y1" }
    edge: { sourcename: "s12`f22`u2"  targetname: "s12`badm`y1" }
  }
  graph: { title: "s12`f21`u2\n"
    /* list of nodes */
    node: { title: "s12`f21`u2" }
    /* list of edges */
  }
  graph: { title: "s11`f12`u2\ns11`badm`y1\ns11`badm`y2\n"
    /* list of nodes */
    node: { title: "s11`f12`u2" }
```

```
    node: { title: "s11`badm`y1" }
    node: { title: "s11`badm`y2" }
    /* list of edges */
    edge: { sourcename: "s11`badm`y1"  targetname: "s11`f12`u2" }
    edge: { sourcename: "s11`badm`y2"  targetname: "s11`f12`u2" }
    edge: { sourcename: "s11`badm`y1"  targetname: "s11`badm`y2" }
    edge: { sourcename: "s11`f12`u2"  targetname: "s11`badm`y2" }
    edge: { sourcename: "s11`badm`y2"  targetname: "s11`badm`y1" }
    edge: { sourcename: "s11`f12`u2"  targetname: "s11`badm`y1" }
  }
  graph: { title: "s11`f11`u2\n"
    /* list of nodes */
    node: { title: "s11`f11`u2" }
    /* list of edges */
  }
  /* list of edges */
  edge: { sourcename: "s11`badm`y1"  targetname: "s22`f4`u1" }
  edge: { sourcename: "s11`badm`y2"  targetname: "s22`f4`u1" }
  edge: { sourcename: "s21`badm`y1"  targetname: "s22`f4`u1" }
  edge: { sourcename: "s21`badm`y2"  targetname: "s22`f4`u1" }
  edge: { sourcename: "s11`badm`y1"  targetname: "s22`goodm`y2" }
  edge: { sourcename: "s11`badm`y2"  targetname: "s22`goodm`y2" }
  edge: { sourcename: "s21`badm`y1"  targetname: "s22`goodm`y2" }
  edge: { sourcename: "s21`badm`y2"  targetname: "s22`goodm`y2" }
  edge: { sourcename: "s22`f4`u1"  targetname: "s22`goodm`y2" }
  edge: { sourcename: "s22`f4`u2"  targetname: "s22`goodm`y2" }
  edge: { sourcename: "s11`badm`y1"  targetname: "s22`goodm`y1" }
  edge: { sourcename: "s11`badm`y2"  targetname: "s22`goodm`y1" }
  edge: { sourcename: "s21`badm`y1"  targetname: "s22`goodm`y1" }
  edge: { sourcename: "s21`badm`y2"  targetname: "s22`goodm`y1" }
  edge: { sourcename: "s22`f4`u1"  targetname: "s22`goodm`y1" }
  edge: { sourcename: "s22`f4`u2"  targetname: "s22`goodm`y1" }
  edge: { sourcename: "s21`f3`u1"  targetname: "s21`badm`y2" }
  edge: { sourcename: "s21`f3`u2"  targetname: "s21`badm`y2" }
  edge: { sourcename: "s21`f3`u1"  targetname: "s21`badm`y1" }
  edge: { sourcename: "s21`f3`u2"  targetname: "s21`badm`y1" }
  edge: { sourcename: "s11`badm`y1"  targetname: "s12`f21`u2" }
  edge: { sourcename: "s11`badm`y2"  targetname: "s12`f21`u2" }
  edge: { sourcename: "s11`badm`y1"  targetname: "s12`badm`y2" }
  edge: { sourcename: "s11`badm`y2"  targetname: "s12`badm`y2" }
  edge: { sourcename: "s12`f21`u2"  targetname: "s12`badm`y2" }
  edge: { sourcename: "s11`badm`y1"  targetname: "s12`badm`y1" }
  edge: { sourcename: "s11`badm`y2"  targetname: "s12`badm`y1" }
  edge: { sourcename: "s12`f21`u2"  targetname: "s12`badm`y1" }
  edge: { sourcename: "s11`f11`u2"  targetname: "s11`badm`y2" }
  edge: { sourcename: "s11`f11`u2"  targetname: "s11`badm`y1" }
}
```

B Hydro power example

B.1 hydro.omt (ObjectMath code)

Some sensitive numerical data has been replaced with **#?#** in this listing.

```
MODEL hydro;
  (* We want all arguments to conditionals to be evaluated symbolically
     as far as possible *)
  ClearAttributes[If, HoldAll];
  ClearAttributes[Switch, HoldAll];

  (* global constants *)
  AccelerationDueToGravity := 9.81;

  (* literals *)
  EP0; EP1; EP2; EP3;  Yes;   (* regulator switches *)
  t;                          (* time *)

  (* index declaration *)
  Indices := { A, B, C, D, E, F, G, H, K, L, M, N, O, P, Q,    (* gates *)
       W1, W2, W3, W4, W5, W6 };                         (* weirs *)



CLASS Source
  InFlow;  (* virtual *)
END Source;

CLASS Sink
  OutFlow;  (* virtual *)
END Sink;

CLASS UpStreamStation(PropagationDelay,SlopeFactor)
  (* PropagationDelay : time for upstream station's flow rate change to
                        propagate
     SlopeFactor : level difference per m3/s between upstream station's
                   outlet and our inlet
  *)

  SurfaceLevel := StationDam`SurfaceLevel + StationDam`InFlow*SlopeFactor;
  OutFlow;  (* input variable *)
END UpStreamStation;

CLASS Spillway(ThresholdHeight,Width) INHERITS Sink
  LevelAboveThreshold := Max[StationDam`SurfaceLevel - ThresholdHeight, 0];
 (* zero when surface is below threshold *)
  SpillHeight := LevelAboveThreshold;
  SpillCrossSection := SpillHeight*Width;
  FlowVelocity := Sqrt[2*AccelerationDueToGravity*
```

```
    (LevelAboveThreshold-SpillHeight/2)];
  OutFlow := FlowVelocity*SpillCrossSection;
END Spillway;

CLASS SpillwayWeir(ThresholdHeight,Width)
  INHERITS Spillway(ThresholdHeight, Width)
END SpillwayWeir;

CLASS SpillwayGate(ThresholdHeight,Width,GateRadius,HubHeight)
  INHERITS Spillway(ThresholdHeight, Width)
PART GateRegul INHERITS SoftRamp(#?#,#?#,`GateAngleRef,`GateAngle);
  GateAngle;
  GateAngleRef;
  HubAboveThreshold := HubHeight-ThresholdHeight;
  AngleBelowHorizon := ArcSin[HubAboveThreshold/GateRadius]-GateAngle;
  HubXDistance := Sqrt[GateRadius^2 - HubAboveThreshold^2];

  SpillHeight := (* The surface level puts a limit to the spill height *)
                 Max[HubAboveThreshold-HubXDistance*Tan[AngleBelowHorizon],
                     LevelAboveThreshold];
END SpillwayGate;

CLASS Turbine(k1,k2) INHERITS Sink
PART Regul INHERITS TurbineRegulator(`Throttle,`Valve);
PART Dyna INHERITS SoftRamp(5, 30,`Valve,`Throttle);
  NominalFall = 21.5;  (* Nominal height of fall, used for *)
  Valve;        (* Controlled by object regulator *)
  Throttle;     (* Actual throttle value caused by valve opening *)
  ThrottleRef;  (* Input variable, controlled by SurfaceRegulator
   or manually, by ControlRoom *)

  `Regul`Reference := ThrottleRef;

  (* Regul:  (Reference, Throttle, t) => Valve
     Controls the hydraulic valve for the water inlet,
     aiming for a certain throttle percentage.  *)

  (* Dyn:    (Valve, t) => Throttle
     Describes how the valve influences the throttle
     percentage.  Approximated by a smoothed ramp.  *)

  NominalFlow := k1*Throttle + k2*Throttle^2;
  FallAdjustedFlow := NominalFlow*Sqrt[NominalFall/StationDam`SurfaceLevel];

  (* derived classes may intervene here *)

  OutFlow := FallAdjustedFlow;
END Turbine;
```

```
CLASS Francis(k1,k2) INHERITS Turbine(k1, k2)
END Francis;

CLASS Kaplan(k1,k2) INHERITS Turbine(k1, k2)
END Kaplan;

CLASS Regulator(Feedback,Output)
END Regulator;

INSTANCE Canal INHERITS Source
  InFlow := Lanforsen‘Outflow[t-Lanforsen‘PropagationDelay];
END Canal;

CLASS Dam(HFlow, WaveAmp, WaveFreq)
  (* HFlow    : input flow required for increasing the surface level
one centimeter in one hour
     WaveAmp  : Amplitude of simulated waves
     WaveFreq : Frequency of simulated waves
  *)

  HFlow;
  InFlow;         (* virtual *)
  OutFlow;  (* virtual *)

  NetInFlow := InFlow - OutFlow;

  SurfaceLevel;

  Waves := WaveAmplitude*Sin[2*Pi*WaveFreq*t];
  tD[SurfaceLevel] := (NetInFlow/HFlow)/3600/100+D[Waves,t];
END Dam;

INSTANCE StationDam INHERITS Dam(#?#, #?#, #?#)
  InFlow := Canal‘InFlow;
  OutFlow := Turbines‘OutFlow + Spillways‘OutFlow + SalmonLadder‘OutFlow;
END StationDam;

INSTANCE Turbines INHERITS Sink
PART G2 INHERITS Francis(#?#, #?#);
PART G1 INHERITS G3Like;
PART G3 INHERITS G3Like;
PART G5 INHERITS SplittableFrancis(#?#, #?#);
PART G4 INHERITS G3Like;
PART G6 INHERITS Kaplan(#?#, #?#);
  OutFlow :=  ‘G1‘OutFlow +
      ‘G2‘OutFlow +
      ‘G3‘OutFlow +
      ‘G4‘OutFlow +
      ‘G5‘OutFlow +
```

```
      `G6`OutFlow;
END Turbines;

INSTANCE Spillways INHERITS Sink
PART MiddleFallGate[k] INHERITS MiddleFallGate;
PART GreatFallGate[k] INHERITS GreatFallGate;
PART SpillWeir[k] INHERITS SpillWeir;
  OutFlow := `GreatFallGate[A]`OutFlow +
     `MiddleFallGate[B]`OutFlow +
     `MiddleFallGate[C]`OutFlow +
     `MiddleFallGate[D]`OutFlow +
     `MiddleFallGate[E]`OutFlow +
     `MiddleFallGate[F]`OutFlow +
     `MiddleFallGate[G]`OutFlow +
     `GreatFallGate[H]`OutFlow +
     `GreatFallGate[K]`OutFlow +
     `GreatFallGate[L]`OutFlow +
     `GreatFallGate[M]`OutFlow +
     `GreatFallGate[N]`OutFlow +
     `GreatFallGate[O]`OutFlow +
     `GreatFallGate[P]`OutFlow +
     `GreatFallGate[Q]`OutFlow +
     `SpillWeir[W1]`OutFlow +
     `SpillWeir[W2]`OutFlow +
     `SpillWeir[W3]`OutFlow +
     `SpillWeir[W4]`OutFlow +
     `SpillWeir[W5]`OutFlow +
     `SpillWeir[W6]`OutFlow;
END Spillways;

INSTANCE Lanforsen INHERITS UpStreamStation(#?#, #?#)
END Lanforsen;

CLASS GreatFallGate INHERITS SpillwayGate(#?#, #?#, #?#, #?#)
END GreatFallGate;

CLASS MiddleFallGate INHERITS SpillwayGate(#?#, #?#, #?#, #?#)
END MiddleFallGate;

INSTANCE SalmonLadder INHERITS Sink
  OutFlow;  (* Input variable *)
END SalmonLadder;

CLASS PIDRegulator(Feedback,Output) INHERITS Regulator(Feedback, Output)
  Reference;  (* input variable *)

  RegP;  (* virtual *)
  RegI;  (* virtual *)
  RegD;  (* virtual *)
```

```
  Compensation := 0;  (* May be redefined in derived classes *)
  Error := Reference - Feedback + Compensation;

  IPart;
  tD[IPart] := Error;

  Output := RegP*(Error + RegI*IPart - RegD*tD[Feedback]);
END PIDRegulator;

INSTANCE SurfaceRegulator INHERITS
                            PIDRegulator(StationDam`SurfaceLevel,RefOut)
  RefOut;

  RegP := 75;
  RegI := 1/50;
  RegD := 0;

  Reference := ControlRoom`SurfaceReference;
END SurfaceRegulator;

CLASS TurbineRegulator(Feedback,Output)
  INHERITS PIDRegulator(Feedback, Output)
  (* Try to help (in a small way) in case of frequency dips *)
  FrequencyDip := 50 - PowerLine`Frequency;  (* Sweden has 50 Hz mains *)

  RegP;  (* input variable *)
  RegD := 0;

  (* EP controls how much much we will help *)
  Compensation := Switch[EP, EP0, 20*FrequencyDip,
     EP1, 50*FrequencyDip,
     EP2, 200*FrequencyDip,
     EP3, 500*FrequencyDip];

  (* EP3 is used in emergencies - regulate faster *)
  RegI := If[EP === EP3, 1/78, 1/45];
END TurbineRegulator;

CLASS Scenario
  (* Instances of this class set all the "free" variables in other objects
     to a constant or time dependent value, when their "select" method is
     called from the global object.  One and only one scenario should be
     selected for each compilation/test run.  *)

  FreeVars = {Turbines`G5`Split, SalmonLadder`OutFlow, PowerLine`Frequency,
      Turbines`G1`Regul`EP,
      Turbines`G2`Regul`EP,
      Turbines`G3`Regul`EP,
```

```
    Turbines`G4`Regul`EP,
    Turbines`G5`Regul`EP,
    Turbines`G6`Regul`EP,
    Turbines`G1`Regul`RegP,
    Turbines`G2`Regul`RegP,
    Turbines`G3`Regul`RegP,
    Turbines`G4`Regul`RegP,
    Turbines`G5`Regul`RegP,
    Turbines`G6`Regul`RegP,
    Turbines`G1`ThrottleRef,
    Turbines`G2`ThrottleRef,
    Turbines`G3`ThrottleRef,
    Turbines`G4`ThrottleRef,
    Turbines`G5`ThrottleRef,
    Turbines`G6`ThrottleRef,
    Spillways`GreatFallGate[A]`GateAngleRef,
    Spillways`GreatFallGate[H]`GateAngleRef,
    Spillways`GreatFallGate[K]`GateAngleRef,
    Spillways`GreatFallGate[L]`GateAngleRef,
    Spillways`GreatFallGate[M]`GateAngleRef,
    Spillways`GreatFallGate[N]`GateAngleRef,
    Spillways`GreatFallGate[O]`GateAngleRef,
    Spillways`GreatFallGate[P]`GateAngleRef,
    Spillways`GreatFallGate[Q]`GateAngleRef,
    Spillways`MiddleFallGate[B]`GateAngleRef,
    Spillways`MiddleFallGate[C]`GateAngleRef,
    Spillways`MiddleFallGate[D]`GateAngleRef,
    Spillways`MiddleFallGate[E]`GateAngleRef,
    Spillways`MiddleFallGate[F]`GateAngleRef,
    Spillways`MiddleFallGate[G]`GateAngleRef,
    EP0, EP1, EP2, EP3, Yes };

DumpDiffs[] := Block[ {file, rlist},
 file = OpenWrite[ "hydro.diffeq",  FormatType-> TextForm, PageWidth->80];
 Write[ file, Indices];
 Write[ file, FullForm[ { DS[],
  DG[Turbines`G1],DG[Turbines`G2],DG[Turbines`G3],
  DG[Turbines`G4],DG[Turbines`G5],DG[Turbines`G6],
  DGD[Turbines`G1],DGD[Turbines`G2],DGD[Turbines`G3],
  DGD[Turbines`G4],DGD[Turbines`G5],DGD[Turbines`G6],
  GFG[A],GFG[H],GFG[K],GFG[L],GFG[M],GFG[N],
  GFG[O],GFG[P],GFG[Q],
  MFG[B],MFG[C],MFG[D],MFG[E],MFG[F],MFG[G],
  SRF[]}]];
 Close[file];
];

(* utility functions for DumpDiffs[] *)
DS[] := Module[{D}, ( D=StationDam`tD[StationDam`SurfaceLevel];
```

```
    Derivative[1][StationDam`SurfaceLevel] == D )];

  DG[N_] := Module[{D}, ( D=(N=>Dyna`tD[N=>Throttle]);
      Derivative[1][(N=>Throttle)] == D )];

  DGD[N_] := Module[{D}, (
       D=N=>Regul`tD[N=>Regul`IPart];
       Derivative[1][(N=>Regul`IPart)] == D)];

  MFG[N_] :=
    Module[{D}, (
      D=Spillways`MiddleFallGate[N]`GateRegul`tD
 [Spillways`MiddleFallGate[N]`GateAngle];
      Derivative[1][Spillways`MiddleFallGate[N]`GateAngle] == D)];

  GFG[N_] :=
    Module[{D}, (
      D=Spillways`GreatFallGate[N]`GateRegul`tD
 [Spillways`GreatFallGate[N]`GateAngle];
      Derivative[1][Spillways`GreatFallGate[N]`GateAngle] == D)];

  SRF[] := Module[{D}, ( D=SurfaceRegulator`tD[SurfaceRegulator`IPart];
    Derivative[1][SurfaceRegulator`IPart] == D )];


END Scenario;

CLASS SpillWeir INHERITS SpillwayWeir(22.80, 10)
END SpillWeir;

CLASS G3Like INHERITS Francis(1.18, 0)
END G3Like;

CLASS SplittableFrancis(k1,k2) INHERITS Francis(k1, k2)
  (* This turbine can be split, dividing its flow by two *)
  Split;

  OutFlow := If[Split === Yes, FallAdjustedFlow/2, FallAdjustedFlow];
END SplittableFrancis;

CLASS SoftRamp(Slope,Stiffness,InValue,OutValue)
  INHERITS Ramp(Slope, InValue, OutValue)
  Stiffness;
END SoftRamp;

INSTANCE PowerLine
  Frequency;
END PowerLine;
```

```
INSTANCE ControlRoom
  SurfaceReference;
END ControlRoom;

CLASS Ramp(Slope,InValue,OutValue) INHERITS Regulator(InValue, OutValue)
  Stiffness := 100;
  OutValue;
  tD[OutValue] := Max[-Slope, Min[Slope,Stiffness*(InValue-OutValue)]];
END Ramp;

INSTANCE Dummy INHERITS Scenario
  select[] := ( 0;
    Map[FuncGen, FreeVars];
    Turbines`G6`ThrottleRef := SurfaceRegulator`RefOut;
    DumpDiffs[];
  );

  Basename[x_] := If[Head[x] === Symbol, x, Head[x]];

  FuncGen[x_] := (If[x == (ControlRoom`RegTurbine=>ThrottleRef),
     Null, x = (Unique[Basename[x]])[t]]);
END Dummy;
```

B.2 hydro.edge (dependency data)

```
graph: {
  node.borderwidth: 1
  edge.thickness: 1
  /* list of SCC */
  graph: { title: "SurfaceRegulator`IPart\nStationDam`SurfaceLevel\nG6`Throttle\nG6`Regul`IPart\n"
    /* list of nodes */
    node: { title: "SurfaceRegulator`IPart" }
    node: { title: "StationDam`SurfaceLevel" }
    node: { title: "G6`Throttle" }
    node: { title: "G6`Regul`IPart" }
    /* list of edges */
    edge: { sourcename: "StationDam`SurfaceLevel"  targetname: "SurfaceRegulator`IPart" }
    edge: { sourcename: "StationDam`SurfaceLevel"  targetname: "G6`Regul`IPart" }
    edge: { sourcename: "G6`Throttle"  targetname: "G6`Regul`IPart" }
    edge: { sourcename: "SurfaceRegulator`IPart"  targetname: "G6`Regul`IPart" }
    edge: { sourcename: "StationDam`SurfaceLevel"  targetname: "G6`Throttle" }
    edge: { sourcename: "G6`Regul`IPart"  targetname: "G6`Throttle" }
    edge: { sourcename: "SurfaceRegulator`IPart"  targetname: "G6`Throttle" }
    edge: { sourcename: "G6`Throttle"  targetname: "StationDam`SurfaceLevel" }
  }
  graph: { title: "MiddleFallGate`GateAngle$G\n"
    /* list of nodes */
    node: { title: "MiddleFallGate`GateAngle$G" }
    /* list of edges */
```

```
}
graph: { title: "MiddleFallGate‘GateAngle$F\n"
  /* list of nodes */
  node: { title: "MiddleFallGate‘GateAngle$F" }
  /* list of edges */
}
graph: { title: "MiddleFallGate‘GateAngle$E\n"
  /* list of nodes */
  node: { title: "MiddleFallGate‘GateAngle$E" }
  /* list of edges */
}
graph: { title: "MiddleFallGate‘GateAngle$D\n"
  /* list of nodes */
  node: { title: "MiddleFallGate‘GateAngle$D" }
  /* list of edges */
}
graph: { title: "MiddleFallGate‘GateAngle$C\n"
  /* list of nodes */
  node: { title: "MiddleFallGate‘GateAngle$C" }
  /* list of edges */
}
graph: { title: "MiddleFallGate‘GateAngle$B\n"
  /* list of nodes */
  node: { title: "MiddleFallGate‘GateAngle$B" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$Q\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$Q" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$P\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$P" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$O\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$O" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$N\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$N" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$M\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$M" }
```

```
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$L\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$L" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$K\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$K" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$H\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$H" }
  /* list of edges */
}
graph: { title: "GreatFallGate‘GateAngle$A\n"
  /* list of nodes */
  node: { title: "GreatFallGate‘GateAngle$A" }
  /* list of edges */
}
graph: { title: "G5‘Regul‘IPart\nG5‘Throttle\n"
  /* list of nodes */
  node: { title: "G5‘Regul‘IPart" }
  node: { title: "G5‘Throttle" }
  /* list of edges */
  edge: { sourcename: "G5‘Throttle"  targetname: "G5‘Regul‘IPart" }
  edge: { sourcename: "G5‘Regul‘IPart"  targetname: "G5‘Throttle" }
}
graph: { title: "G4‘Regul‘IPart\nG4‘Throttle\n"
  /* list of nodes */
  node: { title: "G4‘Regul‘IPart" }
  node: { title: "G4‘Throttle" }
  /* list of edges */
  edge: { sourcename: "G4‘Throttle"  targetname: "G4‘Regul‘IPart" }
  edge: { sourcename: "G4‘Regul‘IPart"  targetname: "G4‘Throttle" }
}
graph: { title: "G3‘Regul‘IPart\nG3‘Throttle\n"
  /* list of nodes */
  node: { title: "G3‘Regul‘IPart" }
  node: { title: "G3‘Throttle" }
  /* list of edges */
  edge: { sourcename: "G3‘Throttle"  targetname: "G3‘Regul‘IPart" }
  edge: { sourcename: "G3‘Regul‘IPart"  targetname: "G3‘Throttle" }
}
graph: { title: "G2‘Regul‘IPart\nG2‘Throttle\n"
  /* list of nodes */
  node: { title: "G2‘Regul‘IPart" }
```

```
    node: { title: "G2`Throttle" }
    /* list of edges */
    edge: { sourcename: "G2`Throttle"  targetname: "G2`Regul`IPart" }
    edge: { sourcename: "G2`Regul`IPart"  targetname: "G2`Throttle" }
  }
  graph: { title: "G1`Regul`IPart\nG1`Throttle\n"
    /* list of nodes */
    node: { title: "G1`Regul`IPart" }
    node: { title: "G1`Throttle" }
    /* list of edges */
    edge: { sourcename: "G1`Throttle"  targetname: "G1`Regul`IPart" }
    edge: { sourcename: "G1`Regul`IPart"  targetname: "G1`Throttle" }
  }
  /* list of edges */
  edge: { sourcename: "G1`Throttle"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "G2`Throttle"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "G3`Throttle"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "G4`Throttle"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "G5`Throttle"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$A"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$H"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$K"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$L"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$M"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$N"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$O"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$P"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "GreatFallGate`GateAngle$Q"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "MiddleFallGate`GateAngle$B"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "MiddleFallGate`GateAngle$C"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "MiddleFallGate`GateAngle$D"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "MiddleFallGate`GateAngle$E"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "MiddleFallGate`GateAngle$F"  targetname: "StationDam`SurfaceLevel" }
  edge: { sourcename: "MiddleFallGate`GateAngle$G"  targetname: "StationDam`SurfaceLevel" }
}
```

C Roller bearing example

C.1 crb-2D.omt (ObjectMath code)

```
MODEL TwoDimBearing;
  (* mathematic rules *)
    Unprotect[Power];
    ((x_)^2)^(1/2) := Abs[x];
    Protect[Power];

    Unprotect[ArcTan];
    ArcTan[Cos[x_],Sin[x_]] := x;
    ArcTan[ x:(z_*_), y:(z_*_)] := (Sign[z]-1)/2*Pi + ArcTan[x/z,y/z];
```

```
    ArcTan/: Sin[ArcTan[x_,y_]] := y/(x^2+y^2)^(1/2);
    ArcTan/: Cos[ArcTan[x_,y_]] := x/(x^2+y^2)^(1/2);
    Protect[ArcTan];

    Norm[x_] := (x.x)^(1/2);

    CrossProduct[a_,b_]:=
      { a[[2]]*b[[3]]-b[[2]]*a[[3]],
       -a[[1]]*b[[3]]+b[[1]]*a[[3]],
a[[1]]*b[[2]]-b[[1]]*a[[2]]  };


  VCrit = 5*10^-4;
  (*SmoothSign[v_] := ArcTan[v/VCrit]*2/Pi;*)
  SmoothSign[v_,der_] := tanh[v*der];
  SmoothSign[v_] := tanh[v/(VCrit*3)];


  (* language constructs *)
  SetOptions["stdout", PageWidth -> 227 ];

  debug[x_,y_:Infinity] := Timing[ByteCount[TracePrint[x, TraceDepth -> y]]];
  SetAttributes[debug, HoldFirst];

  (*ClearAttributes[If,HoldAll];
  SetAttributes[If,HoldRest];*)

  SDO[a_,b_] := a := a = b;
  SetAttributes[SDO,HoldAll];

  SDOaE[a_,b_,e_] := a := a = e @ b;
  SetAttributes[SDOaE,HoldAll];

  SDOaS[a_,b_] := SDOaE[a,b,Simplify];
  SetAttributes[SDOaS,HoldAll];

  psimplify= { Tanh };

  Unprotect[Evaluate[Apply[Sequence, psimplify]]];

  (*Map[# = *)

  Concat[a_,b_] := Block[{res}, setCt[b];
    res=ToExpression[StringJoin[ToString[a],ToString[b]]]; resetCt; res];

  setCt[a_] := $Context= Context[a];
  resetCt := $Context = "Global`";
```

```
  (* global names *)
  Global;
  ang;
  r1;
  r2;
  F;
  v;
  x;
  y;
  vector;
  point;
  velocity;
  from;
  to;


CLASS SpinningElement
  Origin;
  Turn;
END SpinningElement;

CLASS CoordinateSystem(inst) INHERITS SpinningElement
   (* Transformation matrix from local to global system *)
  X := {  Cos[Turn[[3]]], Sin[Turn[[3]]], 0 };
  Y := { -Sin[Turn[[3]]], Cos[Turn[[3]]], 0 };
  Z := {               0,              0, 1 };

  SDO [TransformationMatrix, Transpose[{ X,Y,Z }]];

  SDO [vOrigin, Dt[Origin,inst`t]];

  TranslateAfter[a_,dir_,type_] :=
    Block[{trans},
      Which[type===vector,trans={0,0,0},
            type===point,trans=Origin,
            type===velocity,trans=vOrigin ];
      If[ dir===from, a+trans, a-Transpose[TransformationMatrix].trans ]
    ];

  (* the system is rotating but the velocities
     are located at the origin only *)

  Transform[a_,dir_,type_] :=
    Block[{mat},
      If[ dir===from,
          mat = TransformationMatrix,
          mat = Transpose[TransformationMatrix]];
      TranslateAfter[mat.a, dir,type]
    ];
```

```
  SetAttributes[Transform, HoldAll];
END CoordinateSystem;

CLASS ContactSystem(body1,body2,contact,inst)
  INHERITS CoordinateSystem(inst)
  (* # Global *)

  SDO [RO, body2‘Origin - body1‘Origin];

  SDO [RO1, body2‘rsign* RO]; (* radius direction vector *)

  SDO [RO2, -body1‘rsign*body2‘rsign* RO]; (* coordinate system
                                              direction vector *)

  SDO [Origin, contact‘CP];

  SDO [Turn, ArcTan[ RO2[[1]], RO2[[2]] ]*{0,0,1}];
    (* first axis pointing toward the body  *)


  SDO [vBodyG, Dt[body1‘Origin,#] ];

  SDO [vRotG, Transform[vRot, from, vector]];

  SDO [vContG, vBodyG + vRotG];

  SDO [vRel, vContG - contact‘vCP];


  SDO [ForceG, Transform[Force, from, vector]];

  SDO [TorqueG, Transform[Torque, from, vector]];



  (* # scalar *)
  SDO [fi, ArcTan[ RO1[[1]], RO1[[2]] ]];


  SDO [sradius, body1‘rsign*body1‘Radius[ fi - body1‘Turn[[3]] ]];

  SDO [lever, sradius + contact‘Delta/2];

  SDO [TorqueRoll, Roll[ contact‘ForceCollision,  vSum[[2]] ]];
```

```
  (* # this,  but fixed *)

  SDO [vRot, CrossProduct[ Dt[body1`Turn,#], {-lever, 0, 0 } ]];

  SDO [vBody, Transform[vBodyG, to, vector]];

  SDO [vCont, vBody +  vRot ];
    (* veliocity at contact point on the body1 *)



  (* # this *)

  SDOaS [vSum, Transform[ contact`vSumG, to, vector]];

  SDOaS [Force, {contact`ForceCollision, contact`ForceSlip, 0 }];

  SDO [TorqueSlip, CrossProduct[ {-lever,0,0} , Force ]];

  SDO [Torque, TorqueSlip + {0,0,TorqueRoll}] ;




  (* internal *)

  (* F vertical force , v horisontal speed , returns torque *)
  Roll[F_, v_] := F*21*10^-7*SmoothSign[v];
END ContactSystem;

CLASS Contact(body1,body2,inst)
PART RoRi INHERITS ContactSystem(body1,body2,this,inst);
PART RiRo INHERITS ContactSystem(body2,body1,this,inst);
  (* # scalar *)

  SDOaS [dist, body1`rsign*body2`rsign*Norm[this`RoRi`RO]];

  SDO [Delta, dist-(this`RoRi`sradius + this`RiRo`sradius)];


  SDO [ForceElastic, Elastic[ Delta, body1`rsign*body1`radius, Infinity ,
                                     body1`rsign*body2`radius, Infinity ]];

  SDO [ForceCollision, SqueezeFactor[ vDelta[[1]] ]*ForceElastic];

  SDO [ForceSlip, Slip[ ForceCollision, vDelta[[2]] ]];
```

```
(* # Global *)

SDO [CP, (this`RiRo`lever*body1`Origin +
          this`RoRi`lever*body2`Origin)/dist];
 (* RoRi`lever + RiRo`lever == dist *)

SDO [vCP, Dt[CP,#]];

SDO [vSumG, this`RoRi`vRel + this`RiRo`vRel];



(* # Global, but moving as RoRi and RiRo *)



(* # RoRi and RiRo *)

SDOaS [vDelta, this`RoRi`vCont + this`RiRo`vCont];
  (* symmetry *)


(* internal *)

Emod = 226*10^9;
(* d gap ,r1,r2 radii,  returns vertical force *)
Elastic[d_,rx1_,ry1_,rx2_,ry2_] :=
  If[d > 0,
    0,
    Evaluate[2^(3/2)/3*Emod/( 1/rx1 + 1/ry1 + 1/rx2 + 1/ry2 )*(-d)^(3/2)]];

CDry = 8*10^-1;
(* v vertical speed, returns approx 1 *)
SqueezeFactor[v_] := 1 -  CDry * SmoothSign[v];

(* F vertical force ( >= 0 ),  v horisontal speed,
   returns horisontal force  *)
Slip[F_, v_]  := -Mu0*SmoothSign[v,gamma]*F;

Mu0 = 1*10^-1;
gamma = 8*10^3;
r = 1*10^-2;
beta = ArcTan[ 1/gamma ];
v2 = Mu0/gamma + r * ( Cos[beta] - Tan[beta]*(1-Sin[beta]));
v1 = v2 - r*Cos[beta];

(* v speed ( >= 0 ), returns friction coefficient ( >= 0 ) *)
Mu[v_] :=
  If[v < v1,
```

```
      Evaluate[gamma*v],
      Evaluate[If[(v < v2),
                  Evaluate[Mu0 + r*(Sin[ArcCos[(v2 - v)/r]] - 1), Mu0]]]];
END Contact;

CLASS Body(inst) INHERITS SpinningElement
  tl = inst`t;

  Origin = { R[#]*Cos[Fi[#]], R[#]*Sin[Fi[#]], 0 };
  R/: Sign[R[#]] := If[R[#]===0,0,1];
  R/: Abs[R[#]] := R[#];
  Turn = {0, 0, T3[#] };

  Radius[ang_] := radius + Avaves*Sin[vaves*ang];

  Mass;
  Inertia = Mass*radius^2/2;

  Force;
  Torque;

  (* internal Body *)

  g = 981*10^-2;
  ForceExternal := { -g*Mass, 0, 0};


  SDO [h1, Evaluate[{ Cos[Fi[#]], Sin[Fi[#]], 0 } .
                    Force/Mass+R[#]*Fi'[#]^2] & ] ;
  SDO [h2, Evaluate[({ Cos[Fi[#]], -Sin[Fi[#]],0 } .
                     Force/Mass-2*Fi'[#]*R'[#])/R[#]] & ];
  SDO [h3, Evaluate[Torque[[3]]/Inertia] & ];

  funs = { R, Fi,  T3 };

  myder := Map[Concat["d",#] &, funs];

  substder := Dispatch[ MapThread[ (#1' -> #2) & , { funs, myder }]];
  substtime := Dispatch[ Map[ (#[tl] -> #) & , lhs ] ];

  SDO [der2, { h1, h2 ,h3 } /. substder ];

  SDO [lhs, Join[funs,myder]];
  SDO [rhs, Join[myder, der2]];


  SDO [DiffEq, MapThread[#1'[tl] == #2[tl] &, { lhs, rhs}]];

  DiffEqDep := StringReplace[ ToString[DiffEq // FullForm],
```

```
                              { StringJoin[ ToString[inst], "`" ] -> "",
                                StringJoin[ "[", ToString[t1], "]"] -> ""} ];

  (*initlist := {0.0349949, 0, 0.83872, 7.9996, 5.6085, 56.0174};*)

  known := Dispatch[Map[#[0] -> 0 &,{dR, Fi, T3}]];

  SDO [initrules,
    Join[known, FindMinimum[ Evaluate[Norm[Map[#[0] &, der2] /. known]^2] ,
                             {R[0],0.034,0.035},
                             {dFi[0],7,8},
                             {dT3[0],56,57},
                             Compiled -> False,
                             WorkingPrecision -> 20] [[2]] ]] ;

  InitialVal := Apply[Equal,initrules,1];

  getval[t_] :=Map[#[t] & , lhs] /. inst`nsolve;
END Body;

CLASS Rings(part,inst) INHERITS Body(inst)
  SDO [Force, inst`Ro`part`RiRo`ForceG  + ForceExternal];

  SDO [Torque, inst`Ro`part`RiRo`TorqueG];
END Rings;

CLASS Roller(inst) INHERITS Body(inst)
PART Or INHERITS Contact(this,inst`Or,inst);
PART Ir INHERITS Contact(this,inst`Ir,inst);
  cIr = inst`Ro`Ir`RoRi;
  cOr = inst`Ro`Or`RoRi;

  SDO [Force, cIr=>ForceG + cOr=>ForceG + ForceExternal];

  SDO [Torque, cIr=>TorqueG + cOr=>TorqueG];

  Radius[ang_] := radius;

  (* indata *)
  radius = 5*10^-3;
  rsign = 1; (* surface out from  center *)

  Mass = 2*10^-3;
END Roller;

CLASS Bearing
PART Ro INHERITS Roller(this);
PART Ir INHERITS Rings(Ir,this);
PART Or INHERITS Rings(Ir,this);
```

```
  t;

  clearance = -10*10^-6;

  this`Ir`Fi[t_] = 0;
  this`Ir`R[t_] = 0;
  this`Ir`T3[t_] = 0;

  this`Ir`radius = 30*10^-3;
  this`Ir`rsign = 1; (* surface out from  center *)
  this`Ir`Avaves = 5*10^-6;
  this`Ir`vaves = 9;

  this`Ir`Mass = Infinity;


  this`Or`Fi[t_] = 0;
  this`Or`R[t_] = 2*10^-6;
  this`Or`T3[t_] = 10*t^2;

  this`Or`radius = (this`Ir`radius + 2*this`Ro`radius + clearance);
  this`Or`rsign = -1; (* surface towards center *)
  this`Or`Avaves = 10*10^-6;
  this`Or`vaves = 5;

  this`Or`Mass = Infinity;

  cd = this`Ro;

  OutFile := (cd=>DiffEqDep // TextForm >> "roller.diffeq.cylind");

  range = {t,0,10};


  SDO [nsolve, NDSolve[ Join[cd=>DiffEq,cd=>InitialVal],cd=>lhs , range,
                        StartingStepSize -> 10^-5, MaxSteps -> 5000,
                        WorkingPrecision -> 20]];
  plot := Map[ Plot[ Evaluate[# //. nsolve ],Evaluate[range],
               PlotRange -> All,
               PlotLabel -> # ] &, Map[#[t] &, cd=>lhs ]] ;
END Bearing;

INSTANCE B INHERITS Bearing
END B;

INSTANCE B2 INHERITS Bearing
  this`Ir`Avaves = 0;
  this`Or`Avaves = 0;
```

```
  this`Or`R[#] = 10^-6;
  this`Or`T3[] = 10*t;
END B2;

INSTANCE B1 INHERITS Bearing
  `Ir`Avaves := 0;
  `Or`Avaves := 0;

  `Or`R[#] := 0;
  `Or`T3[#] := 14*#;
END B1;
```

C.2 crb-2D.edge (dependency data)

```
graph: {
  node.borderwidth: 1
  edge.thickness: 1
  /* list of SCC */
  graph: { title: "B2`Ro`T3\n"
    /* list of nodes */
    node: { title: "B2`Ro`T3" }
    /* list of edges */
  }
  graph: { title: "B2`Ro`dT3\nB2`Ro`R\nB2`Ro`dR\nB2`Ro`Fi\nB2`Ro`dFi\n"
    /* list of nodes */
    node: { title: "B2`Ro`dT3" }
    node: { title: "B2`Ro`R" }
    node: { title: "B2`Ro`dR" }
    node: { title: "B2`Ro`Fi" }
    node: { title: "B2`Ro`dFi" }
    /* list of edges */
    edge: { sourcename: "B2`Ro`R"  targetname: "B2`Ro`dT3" }
    edge: { sourcename: "B2`Ro`dR"  targetname: "B2`Ro`dT3" }
    edge: { sourcename: "B2`Ro`dFi"  targetname: "B2`Ro`dT3" }
    edge: { sourcename: "B2`Ro`R"  targetname: "B2`Ro`dFi" }
    edge: { sourcename: "B2`Ro`dR"  targetname: "B2`Ro`dFi" }
    edge: { sourcename: "B2`Ro`Fi"  targetname: "B2`Ro`dFi" }
    edge: { sourcename: "B2`Ro`dT3"  targetname: "B2`Ro`dFi" }
    edge: { sourcename: "B2`Ro`dFi"  targetname: "B2`Ro`Fi" }
    edge: { sourcename: "B2`Ro`R"  targetname: "B2`Ro`dR" }
    edge: { sourcename: "B2`Ro`Fi"  targetname: "B2`Ro`dR" }
    edge: { sourcename: "B2`Ro`dFi"  targetname: "B2`Ro`dR" }
    edge: { sourcename: "B2`Ro`dT3"  targetname: "B2`Ro`dR" }
    edge: { sourcename: "B2`Ro`dR"  targetname: "B2`Ro`R" }
  }
  /* list of edges */
  edge: { sourcename: "B2`Ro`dT3"  targetname: "B2`Ro`T3" }
}
```