Electronic Notes in Theoretical Computer Science 43 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume43.html 23 pages

The *N*-Body Problem in LOTOS

Kenneth J. Turner

Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA
UK

Abstract

It is shown how the classical *n*-body problem in mechanics can be generalised and formalised in LOTOS. A number of variants are produced by instantiation of the specification framework. These include Newton's cradle, gas motion, the 'game of life', an orrery, a space game, an air traffic simulation and a sailing race. It is shown how these are derived from the generic framework using a configuration tool. The resulting LOTOS specifications are simulated automatically to graphically animate the system behaviour.

1 Introduction

The *n*-body problem is a classical one in mechanics. It considers the motion of a number of masses under gravitational attraction. For two bodies there is an analytical solution. For three or more bodies there is no analytical solution and methods of approximation or simulation must be used. In this paper, the problem is interpreted broadly as the interactions in space over time of a number of interacting elements. The problem is then modelled in LOTOS (Language Of Temporal Ordering Specification [5]) and simulated to obtain the dynamic behaviour as a graphical animation. The reader should briefly preview figure 2 to get a feel for the problem variants to be discussed.

This kind of application takes LOTOS well outside its original domain of protocol specification and analysis. The motivations for exploring the *n*-body problem in LOTOS are:

- The problem offers a significant challenge for LOTOS, stretching its 'envelope' and evaluating its applicability in a completely new area.
- As will be seen, LOTOS can be used to provide a generic formulation. This allows a surprising number of variations on the basic problem to be captured by the same specification framework.

[1] Email: kjt@cs.stir.ac.uk

- A number of the variants to be described have standard approaches, e.g. Lagrangian mechanics for the *n*-body problem or Maxwell-Boltzmann theory for gas motion. However these are treated as separate theories in physics. LOTOS supports a common approach to all the variations.
- As a specification language, LOTOS has formal notions such as equivalence, model-checking and proof. The problem and its variants in physics also have formal models such as Newton's equations of motion or statistical mechanics. However the LOTOS approach is very different, and offers different kinds of analyses. For example, the air traffic model given later might be shown (in principle) to avoid aircraft collisions. It should be said, however, that this is an aspiration of the approach; so far, only simulations have been undertaken.
- The *n*-body problem and its variants offer an entertaining change from the sometimes dry topics studied by formal methods. The author confesses that a significant motivation was simply having fun with the approach!

LOTOS has been used successfully in a number of non-protocol areas. These include aviation [3], computer-integrated manufacturing [8], embedded systems [1], graphics standards [9], hardware [6], medical devices [11], neural networks [4], and visualisation [12]. (Many more references could be given.) As far as the author knows, the *n*-body problem is quite different from any previous application of LOTOS. The only similar work is [13] that can generate graphical animations (the dining philosophers eating).

Section 2 presents the problem framework, showing how the original *n*-body problem can be interpreted in a more general way. A number of variants are illustrated. The LOTOS specification framework to encompass these is then explained. Tool support for simulation is discussed in section 3. Section 4 gives some insight into the variants of the general problem. This is illustrated in sections 5 and 6 for the *n*-body and Brownian motion problems.

2 The Problem in General

2.1 Problem Framework

The system under study consists of a number of interacting elements. In deference to the original problem, the term 'body' is used although its interpretation is quite wide. The bodies interact in a problem-dependent way. For example:

- In the *n*-body problem, interaction is by mutual gravitational attraction. The bodies are considered to 'communicate' their position via the medium of the ether. Although a physicist would find this an odd interpretation, it is nonetheless legitimate.
- In a two-dimensional cellular automaton (Conway's 'game of life'), neighbouring cells interact by exchanging information on their occupancy. Again this can be considered a form of communication.
- In the air traffic model, aircraft interact by communicating their position. This is a more conventional means of communication using radar.

From a LOTOS viewpoint, the system is therefore modelled as a collection of communicating processes (one per body). The 'ether' that permits communication is a common gate. This permits broadcast communication, i.e. information made available by one body (say, its position) becomes known to all other bodies. For a number of the problem variants this is exactly what is needed. For some variants, only information from neighbouring bodies is relevant. The bodies therefore act on information from only nearby bodies. For example:

- For Newton's cradle, only the immediate neighbours of a ball are relevant.
- In the game of life, the destiny of each cell depends only its eight immediate neighbours (orthogonally and diagonally).
- In the sailing race, the rules for resolving conflict apply only when the bodies (yachts, buoys) are within two boat-lengths of each other.

The system is considered to be closed, i.e. to be free from external influences. In LOTOS terms, all communication between bodies is therefore internal (i.e. the *Ether* gate is hidden). In most applications, the bodies act deterministically according to their attributes and the states of other bodies. Some applications, however, permit non-deterministic behaviour:

- In Brownian motion, the particles are subject to collisions with gas molecules. The latter are not modelled, only their effect in causing the particles to move randomly.
- In the space game, space-ships randomly change their heading and launch missiles.
- In the sailing race, the wind randomly changes direction and strength.

For most of the problem variants, the number of bodies is fixed in advance. However, for generality the LOTOS formulation permits bodies to become inactive or to be (re-)activated as the system evolves. This is relevant to the space game, for example, since the space-ships may fire missiles (thus creating new bodies).

Bodies have attributes. For consistency each body has a fixed set of attributes, although bodies use these variously in different problem variants. Figure 1 summarises the use of the following attributes in various applications:

Identity: An identity is simply a body identifier – an integer from 0 upwards.

Kind: The kind of a body identifies it as a mass, a gas molecule, an automaton cell, etc.

Rating: This is a catch-all numerical attribute used for additional information. For example, it gives the mass of a planet or the speed rating of a yacht.

Size: This is used when drawing bodies, but is also relevant to the behaviour of some variants (e.g. the sailing model). It defines the diameter of a planet, for example, or the length of a yacht.

Position, Velocity: These are vectors of arbitrary length, although only one, two and three dimensions apply to the applications studied here.

Heading, Path: These also use vectors. Heading may differ from velocity since, for example, a space-ship or a yacht may point in a different direction to that in which it is currently travelling. Bodies in some problem variants aim to follow

Problem	Kind	Rating	Size	Pos.	Vel.	Head.	Path
N-Body	Mass	mass	diameter	(x,y)	(x,y)	-	-
Pendulum	Weight	suspension	diameter	θ	ω	-	-
Newton's Cradle	Ball	suspension	diameter	θ	ω	-	-
Brownian Motion	Particle	-	diameter	(x,y)	(x,y)	-	-
Gas Motion	Molecule	mass	diameter	(x,y)	(x,y)	-	-
Game of Life	Cell	occupancy	width	(x,y)	-	-	-
Space Game	Missile	mass	diameter	(x,y)	(x,y)	-	-
	Spaceship	mass	length	(x,y)	(x,y)	(x,y)	-
	Star	mass	length	(x,y)	-	-	-
Orrery	Earth, etc.	mass	diameter	(x,y)	(x,y)	-	-
Air Traffic	Aircraft	speed	length	(x,y,z)	(x,y,z)	-	waypoints
	Waypoint	-	diameter	(x,y,z)	-	-	-
Sailing Race	Mark	-	diameter	(x,y)	-	-	-
	Yacht	handicap	length	(x,y)	(x,y)	(x,y)	marks
	Wind	max. speed	arrow	(x,y)	(x,y)	-	-

Fig. 1. Particular Examples of the *N*-Body Problem

a certain path as a list of vectors. A yacht sails a race as a succession of buoys (called 'marks') to be rounded, and an aircraft follows a flight plan that is a list of waypoints to be passed.

Figure 2 suggests how bodies are represented graphically in the different problem variants considered in this paper. The graphical depiction is, of course, outside the LOTOS behaviour simulation. However, it is derived directly from the simulation behaviour. Each step in the system evolution produces a new system state. This is provided to a drawing program that creates the graphical display based on the attributes of each body.

The appearance of a body is not determined directly by the LOTOS specification, although various attributes affect the drawing. For example the size, position and heading of a yacht determine how its hull is drawn. The position of the sail is determined by the yacht's heading and the point from which the wind is blowing. In the air traffic application, the position and flight level of an aircraft are derived from the simulation. In a number of the problem variants, the body identifier can be used by the drawing program to label the body, e.g. the name of a planet or the call-sign of an aircraft.

2.2 Specification Framework

The approach performs a kind of discrete event simulation. System behaviour advances in steps, during each of which the state of all bodies is made known. All

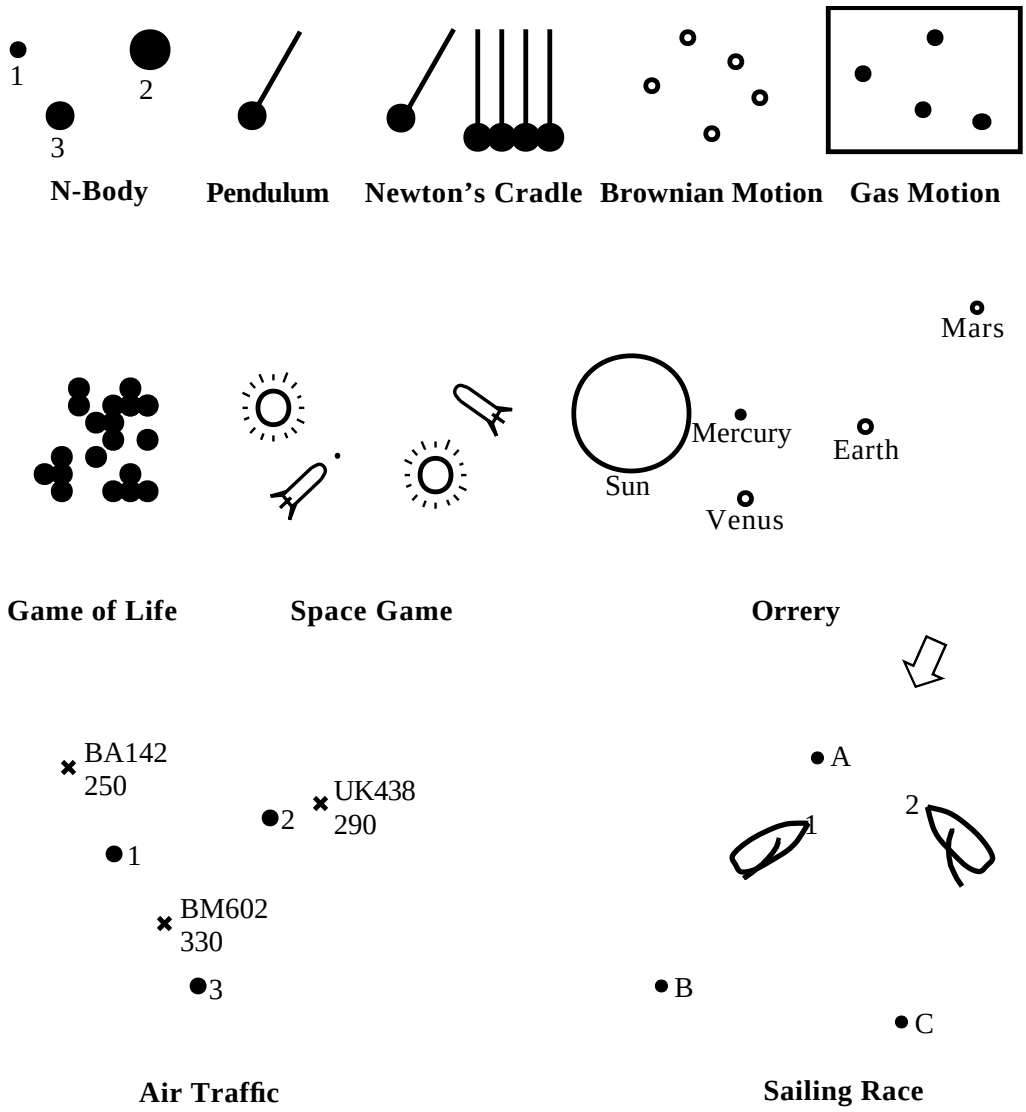

Fig. 2. Particular Examples of the *N*-Body Problem

bodies update then their state, based on their own state and that of other relevant bodies. The next step in system evolution then begins.

The LOTOS specification contains a *Step* process that synchronises the behaviour of all bodies. An *Ether* event is broadcast to all bodies, requesting them to update their state and broadcast this to other bodies. Since all processes synchronise at the *Ether* gate, all events are shared. The processes thus proceed in lock-step. Each body process contains a list of states for other bodies as well as its own state; LOTOS does not support the notion of a global state.

The LOTOS specification framework contains most of what is needed to simulate a variant of the *n*-body problem. Certain 'holes' are left in the specification for completion by a particular application:

any_bodies: contains parallel instances of body processes, with initial values for their attributes.

any_control: is used in a few applications (e.g. the space game or the game of life) that require overall control of bodies. It defines a process rather than a type so that non-deterministic behaviour can be described. It may create new bodies in parallel with existing ones (e.g. missiles in the space game, or new cells in the 'game of life').

any_params: gives any specification parameters or constants.

any_update: defines how to update the state of a body using information about the states of other bodies. This is the heart of the specification, and the part that varies most among problem variants. All the applications define a *StateNext* function:

StateNext : State, States $\Rightarrow$ States

that updates the state for a body using its own state and that of other relevant (e.g. nearby) bodies.

The outline specification framework is as follows. The equations defining types have been omitted for brevity, as have the closing keywords for declarations. The specification uses booleans and strings from the library. Convenient conditional operations are defined for the booleans. Although natural numbers are available from the library, their *Succ* notation is inconvenient. They are therefore implemented by the external C type *int*. Some common constants and operations are introduced for the naturals; *None* is used for an undefined natural. Real numbers are implemented by the external C type *double*. This is particularly necessary since a specification of real numbers would be extremely complex in LOTOS. Some common constants [2] and operations are introduced for the reals. Since LOTOS does not have a built-in syntax for numbers, they are given as modified identifiers (e.g. 3_14, M1_2, 5E3 for 3.14, -1.2, 5×10^3).

The implementation of naturals and reals is given as a parallel description using GLAD (General Language for Annotating Data) that is part of the TOPO toolset [7]. There is a comparable approach for using external types in CADP (Cæsar/Aldébaran Development Package [2]).

```
Specification AnyBody : NoExit                    (* overall specification *)
  Library Boolean, String Endlib                  (* boolean, string *)
  Type BoolOps Is Path, States                    (* boolean operations *)
    Opns                                          (* operations for booleans *)
      If : Bool, Kind, Kind => Kind               (* conditional for kind *)
      If : Bool, Nat, Nat => Nat                  (* conditional for natural *)
      If : Bool, State, State => State            (* conditional for state *)
      If : Bool, States, States => States         (* conditional for states *)
      If : Bool, Path, Path => Path               (* conditional for path *)
  Type Nat Is Boolean                             (* natural number – external *)
    Sorts Nat                                     (* sort name for natural *)
    Opns                                          (* operations for natural *)
```

[2] The definition of *Random* is somewhat naughty since it has the signature of a constant but is implemented by a random function.

```
    0, 1, 2, 3, None : -> Nat                    (* 0, 1, 2, 3, undefined *)
    Succ : Nat -> Nat                            (* successor *)
    _+_, _-_, _*_, _^_ : Nat, Nat -> Nat         (* add/sub/mul/exp *)
    _Eq_, _Ne_, _Lt_, _Le_, _Ge_, _Gt_ :         (* ==/!=/</<=/>=/> *)
      Nat, Nat -> Bool
  Type Real Is Boolean                           (* real number – external *)
    Sorts Real                                   (* sort name for real *)
    Opns                                         (* operations for real *)
      M1, 0, 0_5, 1, 2, Random : -> Real
                                     (* -1.0/0.0/0.5/1.0/2.0/0.0..1.0 *)
      _+_, _-_, _*_, _/_, _^_ : Real, Real -> Real
                                                 (* add/sub/mul/div/exp *)
      Sin, Cos : Real -> Real                    (* sin/cos *)
      Square, Cube, Sqrt : Real -> Real          (* square/cube/square root *)
      Abs : Real -> Real                         (* absolute *)
      _Lt_, _Le_, _Eq_, _Ne_, _Ge_, _Gt_ :       (* </<=/==/!=/>=/> *)
        Real, Real -> Bool
```

Animation is presented using one of three drawing modes: *Label* (body shape plus label), *Shape* (body shape only) and *Trail* (trace of a body's position). The kind of a body is simply an enumerated type that covers all the problem variants. Adding a new kind of body requires a simple extension to the *Kind* type, plus adding new code to the C support library for drawing the body.

```
  Type Mode Is Nat                               (* drawing mode *)
    Opns                                         (* operations for mode *)
      Label, Shape, Trail : -> Nat               (* drawing modes *)
  Type Kind Is Nat                               (* kind of bodies *)
    Sorts Kind                                   (* sort name for kind *)
    Opns                                         (* operations for kind *)
      Aircraft, Ball, Cell, Earth, Mark, Mars, Mass, Mercury, Missile,
        Molecule, Moon, Particle, Ship, Star, Sun, Venus, Waypoint,
          Weight, Wind, Yacht, None : -> Kind
      Ord : Kind -> Nat                          (* ordinal number for kind *)
    _Eq_, _Ne_: Kind, Kind -> Bool               (* (in)equality of kind *)
```

A number of variants on lists are required. These are instantiations of the library string type. The '+' operation for element concatenation is highly overloaded in the library and is renamed '&'. Basic lists are extended with some convenient operations.

```
  Type List Is String RenamedBy                  (* string as list *)
    OpnNames & For +                             (* element prefix/append *)
  Type ListOps Is Nat, List                      (* list operations *)
    Opns                                         (* operations for lists *)
      IsIn : Element, String -> Bool             (* element is in string? *)
      NotIn : Element, String -> Bool            (* element is not in string? *)
```

```
    1st, 2nd, 3rd : String -> Element      (* first/second/third values *)
    Nth : Nat, String -> Element           (* nth string element (0, ...) *)
    Tail : String -> String                (* all but first value *)
```

Nearly all the problem variants require vectors, so these are created as lists of reals with obvious operations. A path is a list of vectors.

```
  Type Vec Is ListOps                      (* vector as number list *)
   ActualizedBy Real, Boolean Using        (* use actual types *)
    SortNames                              (* actual sort names *)
      Real For Element                     (* real as element type *)
      Vec For String                       (* vector sort name *)
      Bool For FBool                       (* use actual booleans *)
    OpnNames                               (* actual operation names *)
      Vec For String                       (* element to vector *)
  Type VecOps Is Nat, Vec                  (* vector operations *)
    Opns                                   (* operations for vectors *)
      _+_, _-_, _*_, _/_: Vec, Vec -> Vec
                                           (* vector-vector add/sub/mul/div *)
      _+_, _-_, _*_, _/_: Real, Vec -> Vec
                                           (* scalar-vector add/sub/mul/div *)
      Mag, SumSquares : Vec -> Real        (* magnitude/sum of squares *)
  Type Path Is ListOps                     (* path as vector list *)
   ActualizedBy Vec, Boolean Using         (* use actual types *)
    SortNames                              (* actual sort names *)
      Vec For Element                      (* vector as element type *)
      Path For String                      (* path sort name *)
      Bool For FBool                       (* use actual booleans *)
    OpnNames                               (* actual operation names *)
      Path For String                      (* element to path *)
```

A state is simply a record-like structure containing the attributes given earlier. A list of states and an operation to update a state are then defined. *NoState* is introduced as the undefined state. The reference to ***any_params*** includes any specification parameters (such as masses or diameters) that are generated for the problem variant. The reference to ***any_update*** includes the problem-defined update procedure.

```
  Type State Is Nat, Kind, Real, Vec, Path (* body state *)
    Sorts State                            (* sort name for state *)
    Opns                                   (* operations for state *)
      State : Nat, Kind, Real, Real,       (* id/kind/rating/size *)
        Vec, Vec, Vec, Path -> State       (* position/velocity/heading/path *)
      Id : State -> Nat                    (* identifier *)
      Kind : State -> Kind                 (* kind *)
      Rate : State -> Real                 (* rating *)
      Size : State -> Real                 (* size *)
```

```
        Pos, Vel, Head : State -> Vec        (* position/velocity/heading *)
        Path : State -> Path                                  (* path *)
        NoState : -> State                          (* undefined state *)
      _Eq_, _Ne_: State, State -> Bool      (* (in)equality for states *)
    Type States Is ListOps                      (* states as state list *)
     ActualizedBy State, Boolean Using          (* use actual types *)
      SortNames                                 (* actual sort names *)
        State For Element                       (* state as element type *)
        States For String                       (* state list sort name *)
        Bool For FBool                          (* use actual booleans *)
      OpnNames                                  (* actual operation names *)
        States For String                       (* element to state list *)
    Type StatesOps Is BoolOps, VecOps           (* state list operations *)
      Opns                                      (* operations for state list *)
        StateUpdate : State, States -> States   (* modify state for body *)
  any_params                                    (* parameters/constants *)
  any_update                                    (* update procedure *)
```

The overall specification behaviour is that of the *Bodies* process synchronised with the *Step* control process. All bodies broadcast their state via the *Ether* gate. Each body may also communicate individually with step control via the *Control* gate. This is used to assign each body a unique identifier when it starts up.

```
    Behaviour                                   (* overall behaviour *)
      Hide Ether, Control In                    (* communication internal *)
        Bodies [Ether, Control]                           (* bodies *)
      ||                                        (* synchronised with *)
        Step [Ether, Control]                   (* start and run simulation *)
          (0 of Nat, 0 of Nat, 0 of Nat, 0 of Real)
    Where                                       (* subsidiary definitions *)
```

The bodies are initialised using the problem-defined ***any_bodies***. This instantiates the ***Body*** process for each body in parallel. The parameters are the attributes for bodies given earlier. In the few problem variants that require it, ***any_bodies*** also includes ***any_control*** for overall body control.

```
    Process Bodies [Ether, Control] : NoExit := (* create bodies initially *)
  any_bodies
```

Initially a body claims a unique identifier by synchronising with the *Step* control process. Bodies are normally initialised with no knowledge of other body states, but if a body is created dynamically during simulation then this information is supplied to it. In such a case the body immediately enters its updating phase *BodyUpdate*, otherwise it enters its waiting phase *BodyWait*.

In the waiting phase, a body is triggered by an *Ether* event to compute and broadcast its new state. The number of active bodies is supplied in this event to all bodies, although this information is not used in most problem variants. If a body becomes inactive during simulation (e.g. a cell in the 'game of life' dies), its kind

is set to *None*. It therefore has no state to update and remains in the waiting phase. Normally the body is active and enters the update phase. While waiting a body may learn the state of other bodies, broadcast over the *Ether* gate. An inactive body may be activated by broadcasting its new state, so there is a check for a matching identifier in the new state. In the updating phase, the body may broadcast its own state and then go back to waiting. It may also learn the state of another body while its state broadcast is pending.

```
Process Body [Ether, Control]                    (* set up body id, states *)
 (kind : Kind, rate, size : Real, pos, vel, head : Vec, path : Path,
  states : States) :
  NoExit :=
  Control ? id : Nat;                             (* get unique body id *)
  (
    Let state : State =                           (* set initial state *)
     State (id, kind, rate, size, pos, vel, head, path) In
      (
        [states Ne <>] ->                         (* other states known? *)
          BodyUpdate [Ether] (state, states)      (* start body states *)
      []                                          (* or *)
        [states Eq <>] ->                         (* other states unknown *)
          BodyWait [Ether] (state, states)        (* start body with states *)
      )
  )
Process BodyWait [Ether]                          (* body waiting phase *)
 (state : State, states : States) : NoExit :=
  Ether ? active : Nat;                           (* get request for updates *)
  (
    [Kind (state) Ne None] ->                     (* body active? *)
      BodyUpdate [Ether]                          (* broadcast new state *)
       (StateNext (state, states), states)
  []                                              (* or *)
    [Kind (state) Eq None] ->                     (* body inactive? *)
      BodyWait [Ether] (state, states)            (* do not broadcast state *)
  )
[]                                                (* or *)
  Ether ? new_state : State;                      (* learn body state *)
  (
    [Id (new_state) Ne Id (state)] ->             (* for other body? *)
      BodyWait [Ether]                            (* repeat with other's new state *)
       (state, StateUpdate (new_state, states))
  []                                              (* or *)
    [Id (new_state) Eq Id (state)] ->             (* for same body? *)
      BodyWait [Ether] (new_state, states)        (* repeat, own new state *)
  )
```

```
Process BodyUpdate [Ether]                          (* body updating phase *)
 (state : State, states : States) : NoExit :=
  Ether ! state;                                    (* broadcast own new state *)
  BodyWait [Ether] (state, states)                  (* wait for new updates request *)
[]                                                                    (* or *)
  Ether ? new_state : State;                        (* learn other body state *)
  BodyUpdate [Ether]                                (* repeat with own new state *)
   (state, StateUpdate (new_state, states))
```

The *Step* process is parameterised by the number of active bodies, the next new body identifier, the count of simulation cycles, and the simulation time. Unless simulation has yet reached the cycle limit (defined as a specification parameter), *Step* can allocate a body identifier on request at the *Control* gate. If there are one or more one active bodies, *Step* can trigger them to update their states using an event at the *Ether* gate. The TOPO annotation (*| C ... |*) is used to call code written in C. Here the function *draw_sync* is called with process parameter 4 (*time*) to draw the state of all bodies on the graphical display. The *Step* process may also be informed of a new body state by broadcast over the *Ether* gate. This special situation arises only if an inactive body is re-activated or if a new body is created dynamically. The C annotation after this event passes the body state (event parameter 1) to the *draw_state* function. This simply records the body state for later drawing when *draw_sync* is called.

The *StepUpdate* process accepts the broadcast of an updated body state, calling *draw_state* to note this. If the new body kind is *None* (because it has become inactive), the count of active bodies is decreased. Normally number of active bodies remains the same, and *StepUpdate* increments the count of bodies it has seen. If more body updates are expected, *StepUpdate* repeats its behaviour. When all updates have been received, behaviour reverts to the main *Step* process.

```
Process Step [Ether, Control]                       (* run simulation steps *)
 (active, nextid, cycles : Nat, time : Real) : NoExit :=
  [cycles Le Limit] ->                                          (* cycles left? *)
    (
      Control ! nextid;                                 (* give new body id *)
      Step [Ether, Control]                     (* repeat with new active, id *)
       (active + 1, nextid + 1, cycles, time)
    []                                                                (* or *)
      [nextid Gt 0] ->                                  (* at least one body? *)
        Ether ! active        (* issue updates request and draw bodies *)
         (*| C draw_sync ($4); |*);
        StepUpdate [Ether, Control]                 (* allow state updates *)
         (0 Of Nat, active, nextid, cycles, time)
    []                                                                (* or *)
      Ether ? state : State           (* note re-activated/new body state *)
       (*| C draw_state ($!1); |*);
      (
```

```
            Let active : Nat =                    (* increment if re-activated *)
             If (active Lt nextid, active + 1, active) In
              Step [Ether, Control]               (* repeat with new active *)
               (active, nextid, cycles, time)
        )
      )
  []                                                              (* or *)
    [cycles Gt Limit] ->                          (* cycle limit reached? *)
      Stop                                          (* finish simulation *)
  Process StepUpdate [Ether, Control]              (* allow state updates *)
   (count, active, nextid, cycles : Nat, time : Real) : NoExit :=
    Ether ? state : State                         (* note new body state *)
     (*| C draw_state ($!1); |*);
    (
      Let                                         (* modify active and count *)
       active : Nat = If (Kind (state) Ne None, active, active − 1),
       count : Nat = If (Kind (state) Ne None, count + 1, count) In
        (
          [count Lt active] ->                    (* more bodies for updating? *)
            StepUpdate [Ether, Control]           (* repeat updates *)
             (count, active, nextid, cycles, time)
        []                                                        (* or *)
          [count Eq active] ->                    (* all bodies updated? *)
            Step [Ether, Control]                 (* repeat step *)
             (active, nextid, cycles + 1, time + Tick)
        )
    )
EndSpec (* AnyBody *)
```

3 Simulation Support

Figure 3 shows the overall flow of information among the tools. The LOTOS specification can be executed using a standard simulator such as the LOLA (LOTOS Laboratory) simulator from TOPO, or alternatively the CADP simulator. This can be done manually, either step-by-step or automatically according to some criterion such as search depth. However the aim is to run the simulation automatically. The TOPO LOTOS-to-C compiler is therefore used to generate fully executable code. The compiled LOTOS is supplemented by implementations of naturals and reals, and also by an interface to the graphical display. The approach is essentially platform-independent. However graphical display necessarily varies as it depends on the supporting system. In the author's environment (OPENSTEP), graphics are handled by a Display PostScript interpreter. Bodies are drawn using PostScript, which is fairly widely supported. Normally each animation frame is sent to the screen, but the drawing functions can also divert each frame to a file. This can be

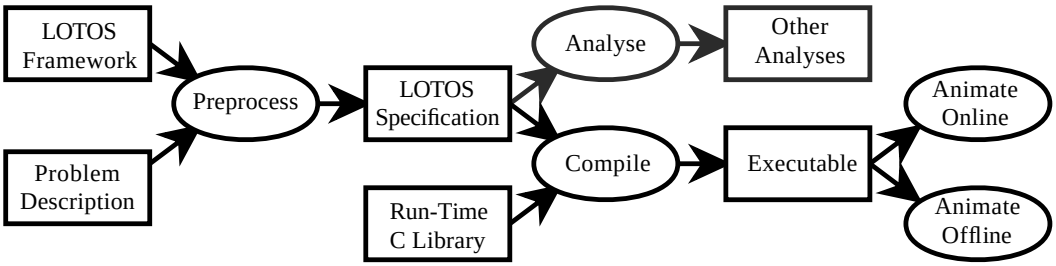

Fig. 3. Toolset Support

converted to an animated GIF (Graphics Interchange Format) file using the public domain *pstogif* and *whirlgif* tools.

The specification framework in section 2.2 needs to be instantiated for the various problem variants. For convenience, the framework is held as a single template that is filled in using a macro processor – *m4* [10] in this case. This allows one specification framework to be used for all the variants. Since the goal is graphical simulation, it would be preferable to enter the initial configuration of bodies graphically. This is quite feasible, but has not yet been done. The general features of *m4* (such as **define** for a macro) may also be used in the problem description. For example, the *n*-body macro library includes *randint* and *randreal* for random integers and reals.

A small number of macros are pre-defined: **body** (body attributes), **control** (control procedure), **param** (specification parameter), **update** (update procedure), and **vec** (vector value). Parameters for **body** may be omitted, and default to sensible values. For example the velocity, heading and path parameters are not used for cells in the 'game of life', and default to empty lists. The various macros accumulate information about the bodies and their attributes. Finally, **update** inserts this information into the specification framework and generates the complete LOTOS file.

The specification parameters are mainly used to control the simulation. Various parameters are pre-defined and may be re-defined if necessary. These include *Limit* (number of simulation steps), *Mode* (drawing mode), *Pause* (time between animation frames), *Scale* (drawing scale), and *Tick* (simulation time step). Other specification parameters control only how bodies are drawn (e.g. the *Diameter* of a planetary body or particle). Parameters controlling the specification may also be defined (e.g. the *Impulse* delivered to a particle in Brownian motion). The parameters, plus other numerical constants, are accumulated in ***any_params*** for inclusion in the final specification.

The LOTOS specification is essentially complete and can be used for other analyses such as verification and validation. The only external support is for numbers. Although this 'cheats' slightly, numbers have well-known theories that can be used for more formal analysis. The implementation of numbers makes use of a GLAD file that describes how to implement data. Although GLAD annotations can be given directly as special comments in the LOTOS specification, this tends to clutter up the specification. A separate GLAD file is therefore provided. As an example,

here is a fragment of how real numbers are implemented:

```
Specification (* Real *) ⇛                          (* implementation of reals *)
        (*| ldc #include "real.h" |*)                  (* header file for reals *)
        (*| ldcinit Real = ud_sort ("Real"); |*);(* initial code for reals *)
    Sorts
      Real ⇛                                        (* basic functions for reals *)
        (*| extern |*) (*| draw Real_Draw |*) (*| equal Real_Check |*)
        (*| free Real_Free |*) (*| parse Real_Parse |*);
    Opns
      Any : forall ⇛ Real ⇛    (* prefix for functions producing reals *)
        (*| using Real |*);
      1 : ⇛ Real ⇛                          (* implementation of constant 1 *)
        (*| constructor |*) (*| extern |*) (*| call Real_Make (1.0) |*);
      _+_ : Real, Real ⇛ Real ⇛   (* implementation of real addition *)
        (*| extern |*) (*| name Add |*);
EndSpec (* Real *);
```

This requires the C header file *real.h* and initialises the sort code for reals. An external sort must be provided with functions to draw (print) values of the sort, check for equality, free allocated space, and parse textual values of the sort. To avoid clashes of overlead function names in the generated C, the **using** pragma can be used to define a prefix for the C functions (here *Real* for any operation that yields a real). The constant 1 is implemented by calling the function *Real_Make* with 1.0 as argument. Real addition calls the function *Real_Add*. These functions are defined using standard C (with some additional macro definitions and standard libraries). For example, addition of two real numbers (of type *udatum* – user datum) is carried out by:

```
udatum Real_Add (udatum r1, udatum r2) {      /* add reals r1 and r2 */
  udatum r;                                          /* result is r */
  SETREAL (r);                                 /* allocate space for r */
  REAL (r) = REAL (r1) + REAL (r2);          /* perform the addition */
  FREEREAL (r1); FREEREAL (r2);  /* free the space for r1 and r2 */
  return r;                                     /* return the result */
}
```

4 The Problem in Particular

Section 2 has presented the overall strategy for simulating the *n*-body problem and its variants. The following notes explain how the specification framework is instantiated in each case. An informal rather than LOTOS explanation is given. Sections 5 and 6 give details of the *n*-body and brownian motion applications.

The *N*-Body Problem: There is only one kind of body, the mass. Masses are initialised with values for their mass, diameter, position and velocity. To update the state of a mass *m*, its distance *d* to each other mass m_i is calculated. The

gravitational force acting on the mass is then given by $G\frac{mm_i}{d^2}$ where *G* is the gravitational constant. The force is a vector quantity directed towards the other mass. The force vectors for all masses are summed and divided by the mass *m*. This gives the instantaneous acceleration due to gravitational attraction as a vector quantity. The simulation time-step uses the acceleration to determine the new position and velocity. The masses often move in surprisingly complex ways.

Orrery: An orrery is a simulation of the solar system. This is just the *n*-body problem with particular values for the inner planets as bodies (and, in principle, for all planets, planetoids and planetesimals). The bodies are initialised with the relevant mass, diameter, distance from the sun, and orbital velocity. The body kinds generate the labels during drawing.

Space Game: The inspiration for this example is an early computer game. It is rather similar to the *n*-body problem. There are two stars, considered fixed because of their comparatively high mass and separation. There are two space-ships that, in the original game, are controlled by users. However since the LOTOS model is closed, the effect of user control is simulated by strategic changes in the velocity and heading of each space-ship. Each space-ship may also randomly launch a missile; this is given a fixed velocity relative to the heading of the space-ship. The space-ships and the missiles are governed by the gravitational attraction of the stars, as in the *n*-body problem. If a collides with another body, the simulation stops and the remaining space-ship wins (unless it collides with the other space-ship).

Pendulum: There is only one kind of body, a weight, and only one of these. The vectors are one-dimensional as there is only one degree of freedom. The pendulum weight is characterised by its off-vertical angle θ and its angular velocity ω. The pendulum is initialised with a mass, length for the suspension, and initial angle; the initial velocity is zero. For mass *m* and acceleration due to gravity *g*, the tangential force on the weight is $mgsin\theta$. This gives the instantaneous acceleration as a vector with magnitude $gsin\theta$. The simulation uses this to determine the new angular position and velocity. The pendulum weight therefore moves, and in fact oscillates indefinitely since the system is assumed to be lossless.

Newton's Cradle: This is a development of the pendulum. A number of balls are suspended by V-shaped threads that allow them to swing in one plane. When the end ball collides with its neighbour, it stops but its momentum is passed on. In the simulation, the velocity of the originally moving ball is passed to its neighbour. This is then considered to collide with the next ball, and so on down the line. Finally, the outermost ball is launched on its arc with this velocity. The specification correctly describes other situations such as several balls on each side being in motion.

Brownian Motion: In this example there are multiple instances of one kind of body, the particle. These are initialised with values for their diameter and position; velocities are initially zero. The weight of a particle is disregarded since the effect of gravity is considered negligible. The state update process simply adds

a random amount to the velocity of the particle, simulating the effect of collision with a fluid molecule. Particle motion is subject to viscous drag. Over time, each particle follows a random walk.

Gas Motion: This is similar to Brownian motion. There are gas molecules confined within a box. Drag is not relevant. If a molecule collides with another or with the edge of the box, it bounces off elastically.

Game of Life: This example of a cellular automaton was defined by J. H. Conway. There is a single kind of body, the cell. It is initialised with its width (solely for drawing purposes) and a fixed position on a two-dimensional grid. Only occupied cells are modelled (since, the overall space is generally sparse). To update the state of a cell, the occupancy of its orthogonal and diagonal neighbours is checked. An occupied cell with zero or one neighbours is considered to die from loneliness; it becomes vacant. An occupied cell with two or three neighbours remains occupied. An occupied cell with more than three neighbours is considered to die from overcrowding. A vacant cell with three neighbours is born (becomes occupied); it remains vacant in all other cases. From the initial configuration of occupied cells, cells are born or die in complex patterns. The behaviour may reach a static state that does not evolve, may enter a repeated cycle, or may evolve indefinitely.

Air Traffic: At the time of writing, this problem variant has been designed but not yet implemented. The two kinds of body in this system are aircraft and waypoints (fixed points in three-space used to define the path of an aircraft). In practice the wind (e.g. the jetstream) is an important factor in flight, but this is ignored in the model. An aircraft is initialised with its maximum speed, length (for drawing purposes only) and position; its velocity is initially zero. An aircraft is given a list of waypoints as its flight plan – its point of origin, the points along its path, and its destination. The waypoint identifier is used during drawing, while the aircraft identifier is used to generate a fictitious call-sign on the PPI (Plan-Position Indicator – a radar display). The x and y coordinates of the aircraft's position determine its location on the display. The z coordinate dictates how its flight level is displayed (the height in hundreds of feet shown next to the aircraft position).

The state update process is rather crude; a full air traffic simulation would be an immense amount of work. In isolation an aircraft adjusts its velocity in increments, subject to a maximum speed. This maximum decides the size of any changes in velocity. The adjustment is made to send the aircraft towards its next waypoint. When the final waypoint is reached, the aircraft is removed from the simulation. The aircraft also checks its position relative to other aircraft. There is no problem if they are separated by at least 1000 feet vertically, one nautical mile to either side, and five nautical miles ahead. If two aircraft approach closer than this, the upper one climbs. (Clearly this simple rule can lead to conflicts and collisions!)

Sailing Race: At the time of writing, this problem variant has been designed but

not yet implemented. The sailing race resembles the air traffic system but in two dimensions. There are moving yachts and fixed marks (the buoys used to indicate a racing course). The wind is a further kind of 'body'. Unlike the aircraft simulation, yachts have a heading and are subject to the wind. Yachts are normally be initialised to begin near a start line (in practice a line between the starting mark and some fixed position). A yacht completes a course by crossing a finishing line (between the finishing mark and some fixed position).

A mark is drawn as a fixed body. A yacht is drawn according to its length, position and heading. The position of its sail is determined automatically from its heading relative to the wind, varying from close to the centre line of the yacht (when sailing upwind) to extending at right angles to the centre line (when sailing with the wind behind). The wind is drawn according to its position (point of origin) and velocity (which determines the length and direction of the arrow).

This simulation has the most complex state update rules. For a given heading, the speed of a yacht is determined by the relative direction of the wind, the wind strength and the rating of the yacht. Yachts are given a PYH (Portsmouth Yardstick Handicap) rating that is inversely proportional to their estimated speed. The speed of a yacht varies in a complex way according to the angle to the wind. This is approximated by a cardioid function ($r = d(1 + cos\theta)$ in polar coordinates). The speed is used to update the yacht's position during a simulation time-step.

In isolation, a yacht aims to complete the course subject to wind conditions. This is non-trivial because yachts cannot sail much closer to the wind than about 45 degrees. When sailing upwind (the angle between the bearing to the next mark and the wind origin is less than 90 degrees), yachts typically have to tack (zig-zag through 90 degree turns) in order not to sail too close to the wind. Their strategy depends on the relative bearings of the wind and the next mark. When a yacht reaches a mark, this is removed from its path list so that the next mark determines the heading. In pseudo-code, the state update process does the following:

if sailing upwind
  then
    if current tack is more favourable
      then set heading to 45 degrees off the wind
      else tack
  else set heading to next mark

The notion of a favourable tack depends on the wind direction relative to the bearing of the next mark. Basically a yacht aims to sail no further than 45 degrees off the course to an upwind mark.

When a yacht approaches another body, rules defined by ISAF (the International Sailing Federation) dictate which yacht has priority. The full set of rules is complex, so only the most basic are specified. Proximity is taken as being within two boat-lengths of another body (a yacht or a mark). A yacht is said to be on a starboard tack if its sail is out to port (left), or on port tack if its sail is out to starboard (right). Yachts are said to overlap if neither is astern (behind) the other.

An inside yacht is one whose course leads closer a mark. A downwind yacht is further away from where the wind originates.

In pseudo-code, the state update process does the following in the case of conflict:

```
if one yacht is close to a mark
  then
    if the yacht is sailing upwind
      then
        if both yachts are on the same tack
          then the inside yacht has priority
          else the yacht on starboard tack has priority
      else the inside yacht has priority
  else
    if both yachts are on the same tack
      then
        if the yachts overlap
          then the downwind yacht has priority
          else the yacht ahead has priority
      else the yacht on starboard tack has priority
```

The yacht without priority must take avoiding action (e.g. turn away from the other yacht). Clearly there are still possibilities for conflict and collision (which is why the full racing rules are so complex).

5 Sample Application: *N*-Body

The *n*-body application is used to illustrate the overall approach. The problem description gives various simulation parameters:

```
param(Limit, 1500)                    # simulation step limit
param(Mode, Shape)                    # drawing mode
param(Scale, 0.4)                     # drawing scale
```

Body definitions are given as their kind (all *Mass*), mass (in units of 10^{24} kilograms), diameter (in units of 10^6 metres), initial position and velocity (in units of 10^9 metres):

```
body(Mass,  14.0E17,  100.0,  vec(100.0, 550.0),  vec(0.0, 0.0))
body(Mass,  1.2E16,   30.0,   vec(100.0, 950.0),  vec(12.0, 0.0))
body(Mass,  2.1E16,   50.0,   vec(350.0, 250.0),  vec(−9.2, −9.2))
body(Mass,  4.5E16,   60.0,   vec(600.0, 50.0),   vec(−7.0, −7.0))
```

This problem variant computes the next state of a mass by adjusting its position and velocity according to the gravitational acceleration. The new position **p** and velocity **v** depend on the mass of the other body *m*, the distance to the other body *d*, the gravitational acceleration **a**, the gravitational constant *G*, and the time step Δt. As a first approximation, these are related by the formulae:

$$\mathbf{p}' = \mathbf{p} + \Delta t\mathbf{v} + \frac{1}{2}\Delta t\mathbf{a}^2$$
$$\mathbf{v}' = \mathbf{v} + \Delta t\mathbf{a}$$
$$\mathbf{a} = G\frac{m}{|\mathbf{d}|^3}\mathbf{d}$$

However this is inaccurate because the formulae hold only if the acceleration is constant during the time-step. This is reasonable if the masses are distant, but as they approach each other (or if the time step is large) the acceleration changes during the time-step. A better approximation is to allow for half the change in velocity when calculating the new position:

$$\mathbf{p}' = \mathbf{p} + \Delta t\mathbf{v} + \frac{1}{2}\Delta t\mathbf{a} + \frac{1}{2}\Delta t\mathbf{a}^2$$

The problem description gives these formulae in LOTOS terms as the update procedure: [3]

```
update(                                              (* update macro *)
  Type Update Is StatesOps                           (* update procedure *)
    Opns                                             (* operations for updating *)
      Accel : Vec, State, States -> Vec              (* add accelerations for state *)
      StateNext : State, States -> State             (* calculate next state *)
    Eqns
      ForAll
        id : Natural, pos, acc : Vec, mass : Real,
        st, st1, st2 : State, sts, sts1, sts2 : States
        OfSort Vec
          Accel (acc, st, <>) = acc;                 (* use gathered acceleration *)
          Accel (acc, st1, st2 & sts) =              (* add up acceleration *)
            Accel (acc +
              ((BigG * Rate (st2) / Cube (Mag (Pos (st2) - Pos (st1)))) *
                (Pos (st2) - Pos (st1))), st1, sts);
        OfSort State
          StateNext (st, sts) =                      (* next state for mass *)
            State (
              Id (st), Kind (st), Rate (st),         (* same id/kind/rate *)
              Size (st),                             (* same size *)
              Pos (st) +
                (Tick * (Vel (st) + (0_5 * Tick * Accel (<>, st, sts)))) +
                  (0_5 * Square (Tick) * Accel (<>, st, sts)),
              Vel (st) + (Tick * Accel (<>, st, sts)),
              Head (st), Path (st));                 (* same heading/path *)
EndType (* Update *))
```

The problem description is given in the file *nbody.any*. It is translated and simulated by the command **anybody** `nbody`. Figure 4 shows partial screen shots

[3] In general, macro bodies should be given in single quotes to prevent their premature expansion and to protect exposed commas and parentheses.

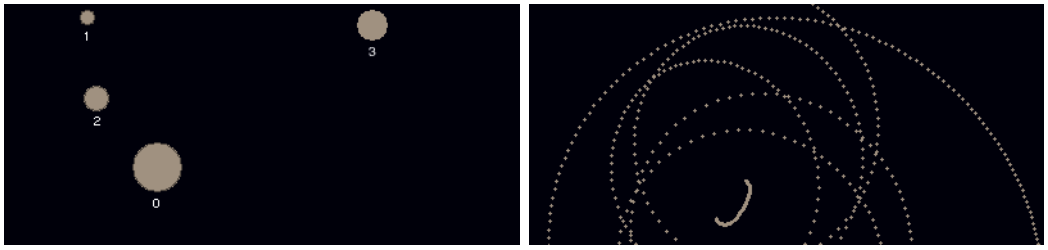

Fig. 4. Partial Screen Shots for *N*-Body Simulation

of the animation. The left-hand diagram shows the *Label* drawing mode, the right-hand diagram shows the *Trail* drawing mode.

6 Sample Application: Brownian Motion

Brownian motion is used as another example of the overall approach. The problem description gives various simulation and specification parameters:

```
define(Diameter, 8)                          # particle diameter
define(PosMax, 300)                          # largest initial position
param(Drag, 0.02)                            # drag coefficient for particle
param(Impulse, 2.)                           # maximum impulse from impact
param(Limit, 600)                            # simulation step limit
param(Mode, Shape)                           # drawing mode
```

The description uses *m4* to instantiate a random number of particle bodies of kind *Particle*, the given particle *Diameter*, and a random position. Macros are defined in *m4* by giving their name and expansion. Macro parameters are identified as $1, etc.

```
define(randpos,                              # calculate random position
  randreal(−PosMax, +PosMax))                # within ± given maximum
define(particles,                            # define given number of particles
  ifelse($1,0,,                              # finish if particle number 0
    body(Particle, 0., Diameter.,            # otherwise instantiate particle body
      vec(randpos, randpos))
    particles(decr($1))))                    # repeat for decremented particle count
particles(randint(20,30))                    # define random number of particles
```

This problem variant computes the next state of a particle by adding a random impulse to its velocity during each time step. The new position **p** and velocity **v** depend on the drag factor *f*, the drag coefficient *d* (based on the particle shape shape and the density of the medium), the random change in velocity $\Delta\mathbf{v}$, and the time step Δt. These are related by the formulae:

$$\mathbf{p}' = \mathbf{p} + \Delta t\mathbf{v}$$
$$\mathbf{v}' = f\mathbf{v} + \Delta\mathbf{v}$$
$$f = 1 - d\Delta t|\mathbf{v}|^2$$

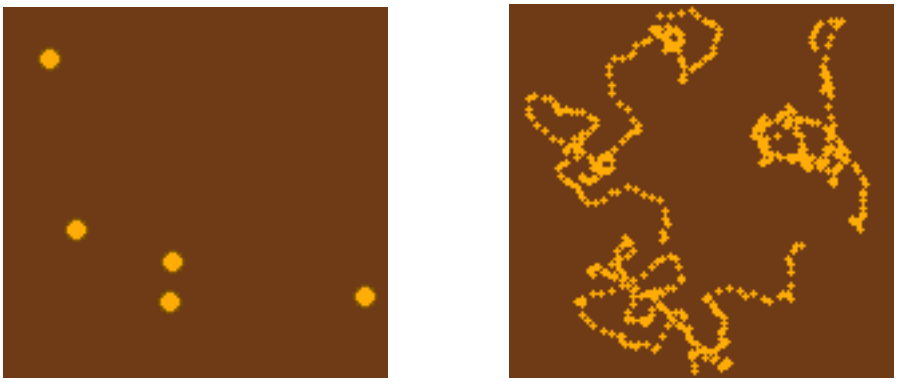

Fig. 5. Partial Screen Shots for Brownian Motion Simulation

The problem description gives these formulae in LOTOS terms as the update procedure:

```
update(                                              (* update macro *)
  Type Update Is StatesOps                           (* update procedure *)
    Opns                                             (* operations for updating *)
      StateNext : State, States -> State             (* calculate next state *)
      DragFactor : State -> Real                     (* calculate drag factor *)
    Eqns                                             (* equations for updating *)
      ForAll st : State, sts : States                (* global state, states variables *)
        OfSort State                                 (* operations that produce a state *)
          StateNext (st, sts) =                      (* next state for particle *)
            State (
              Id (st), Kind (st), Rate (st),         (* same id/kind/rate *)
              Size (st),                             (* same size *)
              Pos (st) + (Tick * Vel (st)),          (* new position *)
              (DragFactor (st) * Vel (st)) +         (* new velocity *)
                (((2*Random - 1) * Impulse) &
                  Vec ((2*Random - 1) * Impulse)),
              Head (st), Path (st))                  (* same heading/path *)
        OfSort Real                                  (* operations that produce a real *)
          DragFactor (st) =
            1 - (Drag * Square (Mag (Vel (st))) * Tick)
EndType (* Update *))
```

The problem description is given in the file *brownian.any*. It is translated and simulated by the command **anybody** `brownian`. Figure 5 shows partial screen shots of the animation display. The left-hand diagram shows the *Shape* drawing mode, the right-hand diagram shows the *Trail* drawing mode.

7 Conclusion

As has been seen, the *n*-body problem can be generalised to an interesting range of applications. Some of these are obvious variations on the original problem, while others are more radical interpretations. Nonetheless it has been possible to develop a specification and simulation framework in LOTOS that encompasses the full gamut of examples. The specification framework is instantiated with the particulars of the kinds of bodies, their initial setup, and how interactions update their states. A pre-processor takes this information and creates a specific instance of the *n*-body problem specification. Standard LOTOS tools (with the help of C annotations) then simulate the behaviour of the system. Each simulation step is animated graphically by a drawing tool, either on-screen or in an animated GIF file.

The *n*-body problem has certainly stretched LOTOS, and has shown its application in a non-traditional domain. An obvious future development is to provide a graphical front-end for initial configuration of objects and subsequent online display of their behaviour. A more challenging objective is to use the formal nature of the specifications to establish properties of the systems. For example an air traffic collision avoidance strategy might be shown to avoid conflicts, or planetary bodies might be shown to orbit without collision.

References

[1] Clark, R. G., *Using LOTOS in the object-based development of embedded systems*, in: C. M. I. Rattray and R. G. Clark, editors, *The Unified Computation Laboratory* (1992), pp. 307–319.

[2] Fernández, J.-C., H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu, *CADP (CÆSAR/ALDÉBARAN Development Package): A protocol validation and verification toolbox*, in: R. Alur and T. A. Henzinger, editors, *Proc. 8th. Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1996 pp. 437–440.

[3] Garavel, H. and R.-P. Hautbois, *Experimenting LOTOS in aerospace industry*, in: T. Rus and C. M. I. Rattray, editors, *Theories and Experience for Real-time system Development*, World Scientific, 1994 .

[4] Gibson, J. P., *A LOTOS-based approach to neural network specification*, Technical Report CSM-112, Department of Computing Science and Mathematics, University of Stirling, UK (1993).

[5] ISO/IEC, "Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour," ISO/IEC 8807, International Organization for Standardization, Geneva, Switzerland, 1989.

[6] Ji He and K. J. Turner, *Specification and verification of synchronous hardware using LOTOS*, in: J. Wu, S. T. Chanson and Q. Gao, editors, *Proc. Formal Methods for*

Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX) (1999), pp. 295–312.

[7] Mañas, J. A., T. de Miguel Moro, T. Robles Valladares, J. Salvachua, G. Huecas and M. Veiga, *TOPO user manual (version 3R6)*, Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain (1995).

[8] McClenaghan, A., *Experience of using LOTOS within the CIM-OSA project*, in: K. R. Parker and G. A. Rose, editors, *Formal Description Techniques IV* (1992), pp. 109–116.

[9] Reade, C. M. P., *Process algebra in the specification of graphics standards*, Technical Report CSTR-92-1, Department of Computer Science, Brunel University, Middlesex, UK (1992).

[10] Seindal, R., *GNU m4 (version 1.4)*, Technical report, Free Software Foundation (1997).

[11] Thomas, M. H., *The story of the Therac-25 in LOTOS*, High Integrity Systems Journal **1** (1994), pp. 3–15.

[12] Turner, K. J., A. McClenaghan and C. Chan, *Specification and animation of reactive systems*, in: V. Atalay, U. Halici, K. İnan, N. Yalabik and A. Yazici, editors, *Proc. International Symposium on Computer and Information Systems XI* (1996), pp. 355–364.

[13] Yasumoto, K., A. Kitajima, T. Higashino and K. Taniguchi, *Hardware synthesis from protocol specifications in LOTOS*, in: S. Budkowski, E. Najm and A. Cavalli, editors, *Proc. Formal Description Techniques XI/Protocol Specification, Testing and Verification XVIII*, Chapman-Hall, London, UK, 1998 .