Appendix J

Dyson

J.1 Introduction

Dyson is a software system for dynamic system design and simulation. Dyson has the following components:

- A dynamic network system description language called ‘DND’.
- A compiler called ‘`dnd2exe`’ (written in Perl) to convert DND into executable C code.
- A simulation kernel (written in C) to simulate the system.

Dyson is much smaller in scope than professional dynamic modeling packages. It currently lacks the ability to form algebraic loops of stateless variables, which means it can not encompass constraint equations between variables. Dyson is intended for dynamical system *design* rather than *modeling*.

J.2 `dnd2exe`

J.2.1 Introduction

`dnd2exe` is a Perl script that take (as standard input) a DND file and produces (as standard output) C code that defines the following functions:

```
void GetSystemInfo (int *nums, int *numd, int *numu, int *numi,
    char **_statenames[], char **_sdnames[], char **_unitnames[],
    char **_maininputnames[]);
void InitializeStateVariables (double *s, double _stepsize);
void InitializeStepDelayUnits (double *d);
void GetStateDerivatives (double t, double stepsize, double *s,
    double *d, double *x, double *in, double *ds);
void GetStepDelayOutputs (double t, double stepsize, double *s,
    double *d, double *x, double *in, double *d2);
void GetStatelessVariables (double t, double stepsize, double *s,
    double *d, double *in, double *x);
void InitializeSystem (double _stepsize);
void ResetSystem (double _stepsize);
```

J.2.2 Simulation external C functions

The simulation functions do the following:

- `GetSystemInfo` : Returns a number of system parameters to the caller:
  - `int nums` : The number of continuous state variables.
  - `int numd` : The number of step delay units.
  - `int numu` : The number of stateless (unit) variables.
  - `int numi` : The number of inputs to the 'main' module.
  - `char *statenames[]` : An array of size `nums` containing the names of the state variables.
  - `char *sdnames[]` : An array of size `numd` containing the names of the step delay units.
  - `char *unitnames[]` : An array of size `numu` containing the names of the stateless (unit) variables.
  - `char *maininputnames[]` : An array of size `numi` containing the names of the inputs to the 'main' module.
- `InitializeStateVariables` : Set the initial values of the state variable array `s`.
- `InitializeStepDelayUnits` : Set the initial values of the step delay unit array `d`.
- `GetStateDerivatives` : Given the time `t`, the state variable array `s` (of size `nums`), the step delay variable array `d` (of size `numd`), the stateless variable array `x` (of size `numu`), and the input array `in` (of size `numi`), return the state derivatives in the array `ds` (which is of size `nums`).
- `GetStepDelayOutputs` : Given the time `t`, the state variable array `s` (of size `nums`), the step delay variable array `d` (of size `numd`), the stateless variable array `x` (of size `numu`), and the input array `in` (of size `numi`), return the new step outputs of the step delay units in the array `d2` (which is of size `numd`).
- `GetStatelessVariables` : Given the time `t`, the state variable array `s` (of size `nums`), the step delay variable array `d` (of size `numd`) and the input array `in` (of size `numi`), return the stateless variables in the array `x` (which is of size `numu`).
- `void InitializeSystem (double _stepsize)` : Initialize stuff which should only be initialized once, i.e. not every time the simulation is reset.
- `void ResetSystem (double _stepsize)` : Reset some simulation stuff between simulation iterations.

J.2.3 Command line arguments

The command line arguments to `dnd2exe` are

- `-p name` : Add the given name as a prefix to all C external function names.
- `-d` : Produce debugging output as a C comment.
- `-i filename` : Set the name of a file to `#include` at the start of the generated C code.
- `-f` : Produce code that uses floats instead of doubles.

J.3 Dynamic network description (DND) language

J.3.1 Introduction

DND is a simple language for high level description of a modular dynamical systems such as a neural networks. With DND you can:

- Specify state variable equations, both continuous and discrete (such as an integrator or first order filter).
- Specify stateless variable equations (where the variable is updated “instantly” at each time step, i.e. there is no time delay).
- Create sub-modules which can form components of larger systems.
- Group variables together and pass the group between modules.

J.3.2 Grammar (in pseudo-BNF)

The DND file is a sequence of tokens that obey the grammar shown here. Comments are written after a ‘#’ character and continue until the end of line. In this grammar character literals are written in upper case, they stand for the string that is the lower case equivalent. Symbol literals are written as they are, e.g. ‘+’.

```
file:
        global_definitions

global_definitions:
        global_definition
        global_definitions global_definition

global_definition:
        function_definition
        group_definition
        module_definition

function_definition:
        FUNCTION name integer [+];

group_definition:
        GROUP name { groupname_list }

groupname_list:
        name
        name = number
        groupname_list , name

module_definition:
        MODULE name ( module_arglist_or_null ) { module_body }
```

```
module_arglist_or_null:
        (null)
        module_arglist

module_arglist:
        name
        name = expression
        group_name name
        module_arglist , name

module_body:
        statement
        module_body statement

statement:
        INT name ;
        INT name ( expression , expression ) ;
        SD name ;
        SD name ( expression , expression ) ;
        UNIT name ;
        UNIT name expression ;
        SUMUNIT name ;
        LINK expression -> name ;
        module_name name ( module_args_1_or_null );
        module_name name [ module_args_2_or_null ];
        group_name name ( group_members ) ;
        UNIQUE name ;
        [INITIAL | RESET] function_name ( comma_expression_or_null ) ;

module_args_1_or_null:
        (null)
        module_args_1

module_args_1:
        expression
        module_args_1 , expression

module_args_2_or_null:
        (null)
        module_args_2

module_args_2:
        name = expression
        module_args_2 , name = expression
```

```
group_members:
        name
        group_members , name

expression:
        or_expression

or_expression:
        xor_expression
        or_expression OR xor_expression

xor_expression:
        and_expression
        xor_expression XOR and_expression

and_expression:
        relop_expression
        and_expression AND relop_expression

relop_expression:
        sum_expression
        relop_expression relop sum_expression

sum_expression:
        product_expression
        sum_expression + product_expression
        sum_expression - product_expression

product_expression:
        unary_expression
        product_expression * unary_expression
        product_expression / unary_expression

unary_expression:
        primary_expression
        + unary_expression
        - unary_expression
        NOT unary_expression

primary_expression:
        name
        number
        ( expression )
        function_name ( comma_expression_or_null )

comma_expression_or_null:
        (null)
        comma_expression

comma_expression
        expression
        comma_expression , expression

name, module_name, group_name:
        Regexp: [A-Za-z_][A-Za-z0-9_.]*

integer:
        Regexp: [0-9]+

number:
        Regexp: [0-9]+([.][0-9]+)?([Ee][+-]?[0-9]+)?

relop:
        <
        >
        <=
        >=
        ==
        !=
```

J.3.3 Expressions

The `and`, `or` and `not` operators work on floating point variables, so they implement the following functions:

- a `and` b = `(a > 0.5) && (b > 0.5)`
- a `or` b = `(a > 0.5) || (b > 0.5)`
- a `xor` b = `(a > 0.5) ^ (b > 0.5)`
- `not` a = `(a <= 0.5)`

The relational operators return 0 or 1.

J.3.4 Statements

This section explains the semantics behind the various syntactical structures shown in the grammar. Later sections will give a fuller explanation of how to use the features of the language.

Global definitions

`function` function-name number-of-parameters [`+`] `;`

Declares that a function (which takes the given number of `double` parameters) has been defined externally (e.g. in the C runtime library). This function can now be used in expressions. If the + is given then the fixed arguments can be followed by any number of variable arguments. In the actual C function call all the arguments are prefixed by an integer which says how many arguments are supplied in the variable section.

`group` groupname { name, name, ... }

Makes 'groupname' a synonym for a group of named variables. This is similar in principle to C's `struct` statement. Default (constant) values can be provided for group elements by saying 'name=number'.

`module` modulename (argument_list) module_body...

Defines a module. The argument list can be empty or it can be a comma separated list of arguments. An argument can be a name followed by an optional '=expression' which gives the default value for the argument. Alternatively the name can be a group name (previously declared with a `group` command) followed by an argument name. See below for how this is interpreted. The module body is a list of statements. Each statement is terminated by a semicolon.

Module statements

`int` name `;` `int` name (derivative_expression, initial_value_expression) `;`

Defines an integrator (state variable) and (optionally) gives expressions for its state derivative and initial value. If no expressions are given then this is a forward declaration, and the integrator must be properly defined later. The initial value expression must evaluate to a constant.

`sd` name `;` `sd` name (next_output_expression, initial_value_expression) `;`

Defines a step delay (discrete state variable) unit and (optionally) gives expressions for its next output and initial value. If no expressions are given then this is a forward declaration, and the step delay unit must be properly defined later. The initial value expression must evaluate to a constant.

`unit` name `;` `unit` name = expression `;`

Defines a stateless variable and (optionally) gives an expression for its output. If no expression is given this is a forward declaration, and the unit must be properly defined later. Algebraic loops in unit expressions are not allowed, so there must not be circular dependencies among units that cannot be resolved without extra equation solving at each iteration.

`sumunit` name type `;`

This defines a unit that can be "linked to". The unit's expression is built using the `link` commands — the unit type determines how the `link` command source expressions are combined to get the unit expression. For sumunits the source expressions are summed together.

module_type module_name (argument_expressions) `;`

This makes an instance of a module 'module_type' called 'module_name'. The module argument names are substituted for the argument expressions given (there must be a one-to-one correspondence). The argument expressions are comma separated.

module_type module_name [argument_assignment_expressions] `;`

This also makes an instance of a module, but here argument assignment expressions can be used, of the form 'name=expression'. The assignment expressions can occur in any order (they do not have to match the declaration order of the module). Module argument names can be repeated, the later assignments will take precedence over the earlier ones (this is a useful mechanism for overriding group arguments). Omitted arguments will take their default values. It is an error if a module argument has neither default value or assignment expression.

group_type group_name (expression_list) `;`

This makes an instance of a group. The comma separated expression list gives the value for each element of the group. The group name can now be used as a synonym for the expression list.

`unique` name `;`

This declares a value that will be unique for each module that it is used in, e.g. if the **unique** name is instanced three times then the three instances will have the values 0,1,2.

`initial` function_name (arguments) `;`

This says that the given function will be executed once, before simulation starts. The arguments should be constant values, but "unique" variables can also be used. As a special case, the `_stepsize` variable can also be used, as it is assumed to remain constant.

`reset` function_name (arguments) `;`

This says that the given function will be executed whenever the simulation is reset. The arguments should be constant values, but "unique" variables can also be used. As a special case, the `_stepsize` variable can also be used, as it is assumed to remain constant.

J.3.5 Modules

A module is a subsystem which can be duplicated multiple times. Within the module new integrators and units can be defined. By default these elements will have local scope, that is their names will not be accessible outside the module. A module has named inputs that can be referenced by expressions within the module.

Module instantiation is similar to macro expansion. When a module is instantiated the module body is substituted for the module command, with the following transformation:

- All module variables are prefixed by the module name followed by a dot.
- All module inputs are substituted for the expressions supplied in the module command.

For example:

```
module foo (a,b,c) {
  int tox (a+b+2, 2-c);
  unit output = b*c;
}
module bar (q) {
  unit r = 2;
  foo dof (q,r*4,1)
  unit pod = dof.tox + dof;
}
```

...makes the module ‘bar’ equivalent to...

```
module bar (q) {
  unit r = 2;
  int dof.tox (q+(r*4)+2, 2-1);
  unit dof.output = (r*4)*1;
  unit pod = dof.tox + dof.output;
}
```

A module called ‘main’ must be defined. The main module is the one that is actually used for simulation. Its inputs are the global inputs to the system and must be supplied by some external agent.

A module can be forward declared by instantiating it without supplying any arguments, e.g.

```
bar b;
unit x = b.pod;
bar b (123);
```

J.3.6 Expressions and variables

All expressions can only refer to the following variables:

- State variables (integrators) or stateless (unit) variables defined locally within the module. Variables must be defined before they are used. As a special case, an integrator expression can refer to itself.
- Module input variables.
- Integrators and units inside module instances, using the dotted qualifying notation. Referring to a module name itself will refer to a variable called ‘name.output’.
- The special variable ‘`t`’ can be used for the current time.
- The special variable ‘`_stepsize`’ can be used for the simulation step size (assuming that it is always the same).

All variables have module local scope.

J.3.7 Groups

Groups allow you to deal with large collections of variables/expressions easily. Groups are implemented using a text-expansion mechanism (a bit similar to macro expansion only smarter). Note that defining a group merely makes the group name a synonym for the grouped variables/expressions, it does not define any new variables.

Suppose we have a group ‘g’:

```
group g {a,b=2}
```

If we use the group as a module argument like this

```
module mymod (p, g foo, q) {...
```

it is equivalent to writing

```
module mymod (p, foo.a, foo.b=2, q) {
  g foo (foo.a, foo.b);
  ...
```

Note that the group arguments get their default values from the `group` declaration (this is the only way that group arguments can have defaults). The group is re-instantiated inside the module so it can still be referred to collectively. If we then pass a group to '`mymod`' inside another module like this

```
g bar (zog,123);
mymod nog [p=1, foo=bar, q=2];
```

it is equivalent to writing

```
mymod nog [p=1, foo.a=zog, foo.b=123, q=2];
```

Now here is a tricky point. The type of '`foo`' in module '`mymod`' is not stored (we just store the arguments `foo.a` etc) so when the group `bar` was passed to the module, *its* type was used to generate the expanded argument list. When this group expansion is done, arguments that are not one of the module's named inputs are omitted. This means that incompatible groups can be passed between modules, as long as enough default argument values exist to patch up the gaps.

When a module is used inside another module, all of its groups are available for reference.

J.3.8 Constant folding

There is not provision to explicitly declare constants, but you can get the same effect by saying

```
unit my_constant_pi = 3.14;
```

A constant folding mechanism will find all references to (and expressions made up of) constant units and replace them by the constant values.