Darwin2K Tutorial

Chris Leger

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Last updated: 6/13/00

1 Introduction

1.1 What is Darwin2K?

Darwin2K is a software package for simulating, synthesizing, and optimizing robots. The motivation behind Darwin2K is to provide a set of software tools to make the robot configuration process easier for the robot designer, and to allow better robots to be designed. Traditionally, the configuration design process (or *configuration synthesis*) has addressed the creation of the high-level form of the robot: whether it will be a manipulator or mobile robot, whether the robot will have wheels or legs or some combination thereof, and so on. After a particular configuration is selected, the dimensions, components, and other specific properties are determined; this may be called parametric design. In some cases, numerical analysis and optimization may be undertaken to determine the best values for specific properties, though often values which simply satisfy a set of requirements are selected by hand.

Darwin2K combines some aspects of the configuration and parametric design processes, for two related reasons. First, the performance and behavior of a configuration often cannot be accurately predicted without specific parametric values. Second, Darwin2K uses simulation to measure a robot's performance, and simulation requires knowledge of the robot's configuration as well as its parametric properties. However, Darwin2K does not address the entire configuration synthesis problem, nor the entire parametric optimization problem. Instead, it tackles the medium- to high-level parametric design and optimization problem, and the low- to medium-level configuration synthesis problem. For example, while Darwin2K can optimize a robot's kinematic dimensions and actuator choices, it will not generate a detailed design with information about cable routing, bolt patterns, and so on. This lowest level of detail for a design can usually be generated from a higher-level design without requiring many design iterations--that is, the higher-level design will not usually require significant modifications to address the issues arising in detailed design.

At the other end of the design spectrum, Darwin2K will not make the highest level of design decisions such as deciding whether a flying, swimming, walking, or rolling robot is best suited for a task. This decision process involves envisioning a task description and accompanying motion plans, which must then be embodied in the choice of robot controller, tool trajectories, and so on, before a configuration's performance can be quantitatively evaluated. The designer can, however, use Darwin2K to generate reasonably detailed, well-optimized designs for multiple categories (e.g. flying or walking) of configurations, allowing the designer to make an informed choice about configuration issues. Simulating a robot requires the designer to provide a description of the task, including properties such as trajectories, controllers, payload models; if the designer can provide these for multiple classes of configurations, then Darwin2K can be used to synthesize and optimize robots for each general class of configurations.

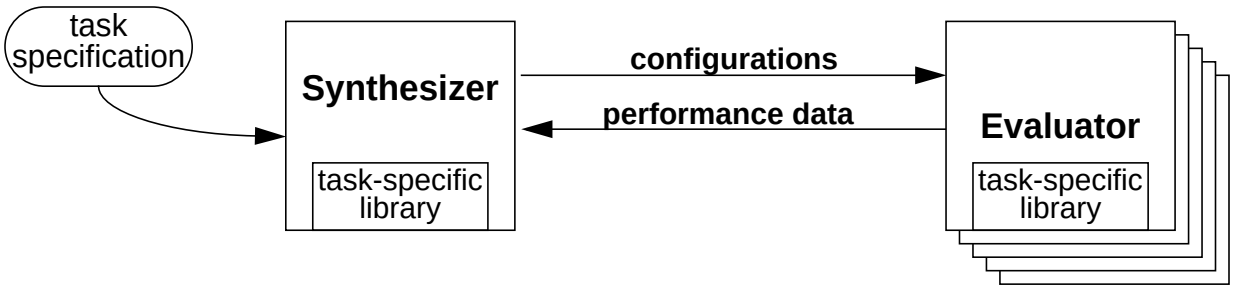


Figure 1.1: Software architecture

Darwin2K consists of two main programs: the synthesizer (or *Population Manager*), and the evaluator. The evaluator includes general-purpose libraries for simulating and controlling robots, while dynamic libraries allow the designer to provided task-specific modules, task descriptions, controllers, and simulation code. Many evaluator processes can be distributed across multiple computers or processors to reduce overall runtime.

1.2 Software overview

Darwin2K contains several programs and a suite of robot simulation libraries. The two main programs are the synthesizer (or *Population Manager (PM)*) and the evaluator (see Figure 1.1). The synthesizer generates robot designs and sends them to the evaluator, which simulates the robot and returns performance measurements to the synthesizer. In practice, many evaluator processes can be distributed across a network of computers (or among multiple CPUs in a multiprocessor computer) since evaluating designs is the most computationally expensive part of the synthesis process. The evaluator program makes use of Darwin2K's libraries of robot parts and simulation and control algorithms. Some of the features included in these libraries are:

- kinematic simulation
- dynamic simulation
- Jacobian-based trajectory following
- PID control
- estimating link deflection
- computing joint torques
- collision detection

The libraries also contain a few dozen *parameterized modules*, which represent robot parts such as links, joints, end effectors, and mobile bases. These modules are the building blocks from which the designer and the synthesizer construct robots.

While a wide range of design problems can be addressed with the capabilities and modules included in Darwin2K, there will always be tasks for which special-purpose con-

trollers or modules are needed. For this reason, Darwin2K uses an extensible, object-oriented software architecture that allows new controllers, simulation algorithms, metrics, robot modules, and other software objects to be added and to interact with existing software components. These objects are written in C++ and can be compiled into dynamic libraries, so that Darwin2K can use them without requiring the evaluator and synthesizer programs to be recompiled.

1.3 Tutorial overview

There are four main parts to this tutorial. In order of increasing complexity, they are:

- specifying robots in Darwin2K;
- simulating robots;
- using the synthesizer; and
- adding task-specific objects.

The first three topics describe the use of Darwin2K and do not require any knowledge of programming, though some information on the C++ implementation will occasionally be given for readers who are interested in extending Darwin2K. The last section (adding task-specific objects) is more involved and assumes a working knowledge of C++ (and of any algorithms the designers might want to implement!). Knowledge of robot kinematics and of basic mechanics is useful but not essential.

2 Building Robot Configurations

2.1 Parameterized Modules

In Darwin2K, robots are built from *parameterized modules*. As the name implies, these modules may have a number of parameters describing their properties. A module that represents a robot joint might have parameters which specify the joint's motor and gearhead, while a module for a wheeled mobile base might have parameters for the suspension type, wheelbase, and wheel diameter. Some parameters represent continuous values such as dimensions, while others are discrete and may, for example, select components from a list. Modules also have *connectors*, which are used to connect multiple modules to form a complete robot. Figure 1.1 shows a `rightAngleJoint` module, which is a joint with one degree-of-freedom (DOF) and which has a 90 degree angle between its two connectors. (Note that the `courier` font is used when discussing software objects, types, or examples; this convention will be followed throughout the tutorial.) There are four general types of modules in Darwin2K (joints, links, bases, and tools), each with special capabilities. For example, joints have degrees of freedom, while tools have tool con-

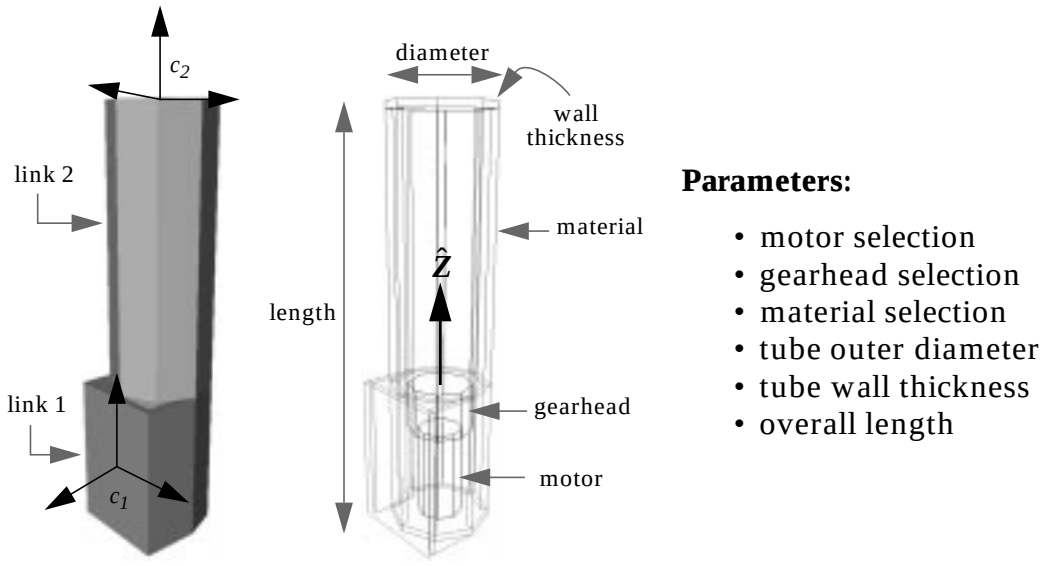


Figure 2.1: Sample parameterized module

A `rightAngleJoint` module is shown here with a list of its parameters. The first two parameters select a motor and gearbox for the module's single degree of freedom (whose joint axis is labelled $\hat{Z}$); one parameter chooses a material (e.g. a particular aluminum alloy) for the module, and the remaining parameters determine the module's size. Each of the parameters can be individually varied by the synthesizer. The module's two connectors are labelled c_1 and c_2, respectively.

trol points (TCPs). Before going into the details of different module types, let us see how modules are specified by the user.

2.1.1 Module names and structure

Darwin2K uses a simple, LISP-like text format for reading and writing modules and configurations. A simple module with no parameters would be described like this:

```
(palletFork nil nil)
```

The token `palletFork` is the module's type; this tells Darwin2K what type of module to create based on the text description--in this case, a set of pallet forks that might be found on a forklift. The first `nil` indicates that the module has no parameters, and the second `nil` indicates that the module has no attachments (i.e. it is not attached to any other modules). Note that the module type (e.g. `palletFork`) cannot be just any string; it has to be the name of an existing module type (actually the C++ class name of the module) in Darwin2K. If you want to see what this module looks like, run the commands

```
% cd $D2KTUTORIAL/examples
% displayCfg palletFork.l
```

(This assumes that you've set the shell environment variable `D2KTUTORIAL` to the directory containing this file, and that the program `displayCfg` is in your path.) If you look at the file `palletFork.l`, you might notice that there is an extra set of parentheses around the module. This is because `displayCfg` expects a complete configuration, not just one module, and a configuration is a list of modules (plus some other stuff we'll get to shortly). So, if you want to use `displayCfg` to look at a single module, remember to enclose the module in an extra set of parentheses (i.e. there should be two left parens before the module name.)

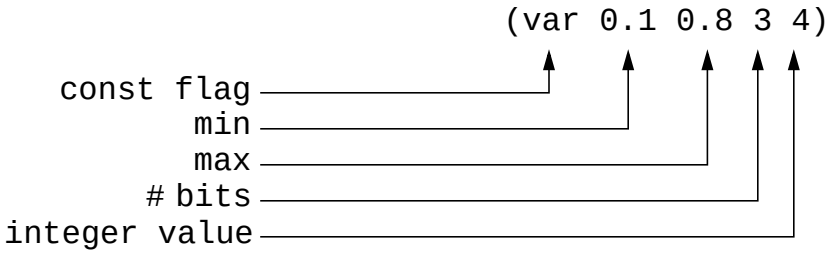
2.1.2 Parameters

A more complicated module can have parameters. One such module is the `squarePrism`, which is a hollow link module that has a square cross-section. The module has two parameters, one for cross-section width and one for length:

```
(squarePrism ((var 0.1 0.8 3 4)
              (var 0.2 1.5 3 5))
             nil)
```

Here, instead of the first `nil` there is a list of two parameters. Note that each parameter

is surrounded by parentheses, as is the list of parameters. Each parameter has five fields:



If we were only concerned with describing the actual value of a parameter (for example, the length) then we would only need one number. However, when the synthesizer is creating or modifying configurations, it also needs to know the minimum and maxium values each parameter can take on, what resolution is needed, and whether or not a parameter's value can be changed. The other parts of the parameter provide this information. The `const`-flag tells the synthesizer whether or not it can change a parameter's value; this is useful if the designer already knows the best value for a parameter, or if there is some sort of constraint on the parameter's value. A parameter's `const`-flag may have one of two values: `var` indicates that the paremeter value is variable, and `const` indicates that it is constant. The next two fields (minimum *m* and maximum *M*) specify the allowable range of parameter values. The fourth field *b* is the number of bits that should be used to represent the parameter's value, and the fifth field *i* is the parameter's integer value, which determines the parameter's actual value (e.g. the length). The actual value *a* is obtained from the number of bits, minimum and maximum values, and integer value by simple linear interpolation:

$$a = m + \frac{i}{2^b - 1}(M - m) \tag{2.1}$$

Values of *i* range from 0 to 2^b-1; thus, an integer value of 0 gives an actual value of *m*, and an integer value of 2^b-1 gives an actual value of *M*. For example, the first parameter of the squarePrism above is

```
(var 0.1 0.8 3 4)
```

and applying Equation 2.1 gives an actual value of 0.5. This may seem overly complicated at first, but if you just want to specify a particular actual value you can set *m* to the actual value you want, pick some value greater than zero for *b*, pick an arbitrary number for *M*, and set *i* to zero. For example if you want a parameter to represent the value 2.5 and you don't care about the resolution or any of that other junk, you can write

```
(var 2.5 2.5 1 0)
```

Since the integer value is 0, the actual value is at the bottom of the range of allowable values and is thus equal to the minimum -- 2.5 in this case.

2.1.3 Selecting components with parameters

(Note: if you're in a hurry to put together a complete robot, go ahead and skip on to Section 2.2).

In the previous example the parameters represented continuous values. But what if a parameter selects a motor from a list of motors? In this case, *m*, *M*, and *b* are ignored and only the const-flag and *i* are used to determine the parameter's value (i.e. to select the motor). While a module type *can* have a hard-coded list of components for its selection parameters, Darwin2K allows each module to specify a *component context*. The component context is simply a list of names of the components that can be used for each of the module's selection parameters. In this way, two modules of the *same type* but with *different* component contexts can represent two very different performance ranges: one joint module might have a component context with very small motors, while another joint module of the same type could have a component context with very large motors. If a module has a component context, it is specified after the module name:

```
(hollowTube materialList1 ((var 0 1 2 0)
                           (var 0.05 1.0 3 2)
                           (var 0.05 0.15 3 7)
                           (var 0.001 0.005 3 0))
                          ())
```

The component context `materialList1` appears after the module name `hollowTube`, indicating that components should be chosen from the list `materialList1`. The hollowTube module is...well, a hollow tube (Figure 2.2). The first parameter is the module's only component selection parameter, and determines which type of material (e.g. aluminum, stainless steel, etc.) the link is made out of. While material isn't exactly a "component", it is a discrete choice.

When a module specifies a component context, Darwin2K has to have a definition for that context--and for the components included in the context--in order to create a physical representation of the module. This information comes from the *component database file*, or `componentDB`. Figure 2.3 shows the part of a component database file that defines the component context for a particular type of joint module. In this case, parameter 0 would select a motor (from the `componentList` labeled "motors"), and parameter 1 would select a gearhead (from the list labeled "gearHeads"); other parameters of the

Parameters:
- material selection
- length
- outer diameter
- wall thickness

Figure 2.2: The `hollowTube` module
The hollowTube is a link module with four parameters. The first parameter (material type) is a selection parameter, and the others are continuous parameters representing dimensions.

module do not select components and so are not mentioned in the component context. In addition to the component contexts, the component database file contains descriptions of each component, such as the density and modulus of elasticity for materials, or the gear ratio, efficiency, and torque and velocity limits for gearheads. Currently, there are components in Darwin2K for electric motors, gearheads, harmonic drives, lead screws, and materials. The motors and gearhead descriptions are taken from Maxon catalogs, while the harmonic drives are from HD Systems. New components of existing types (e.g. motors) can be added by editing the componentDB file, and new types of components (such as hydraulic pumps and actuators) can be added by writing new C++ classes for the components.

Some selection parameters may have dependencies on other selection parameters: for example, one parameter might specify a motor and another might specify a gearhead, but not all gearheads are compatible with all motors. To handle compatibility constraints such as this, Darwin2K allows components to specify dependencies on other components in the component database file. For example, the component description for a particular gearhead is shown in Figure 2.4. The description indicates that the gearhead may only be used with one of two motors, so if a module's motor selection parameter specifies a different motor then this gearhead will not be considered when the module's gearhead selection parameter is being interpreted. The order of a module's parameters determines how the component dependencies are resolved: a component specified by the j^{th} parameter can depend only on the components selected by parameters 0 through j-1, thus preventing cyclic dependencies.

```
context revoluteJoint {
        parameter 0 "motors";
        parameter 1 "gearHeads";
}

componentList "motors" {
        rotaryActuator "maxonRE25.118755";
        rotaryActuator "maxon2260.815";
        rotaryActuator "maxonRE36.118800";
        rotaryActuator "maxon2260.889";
}

componentList "gearHeads" {
        gearBox "maxon16.118188";
        gearBox "maxon26.110396";
        gearBox "maxon32.110464";
        gearBox "maxon42.110404";
}
```

Figure 2.3: Component context and lists

The component context `revoluteJoint` indicates which components lists should be used for a module's selection parameters. The lists themselves (`motors` and `gearHeads` in this example) contain the types and names of specific components, which are also defined in the componentDB file.

To add a new component (of existing type, e.g. gearhead) to the componentDB file, simply create an entry similar to the one shown in Figure 2.4. The first part of the component description is a token for the component's type--`gearBox`. Like the module type in a module description, the component type is a predefined identifier that tells Darwin2K what type of software object to create. The next part of the description is the component's name, written as a string of characters enclosed in quotes. After that is an open brace ('{') starting the list of component properties. The values for each of the component's properties are then given, followed by a close brace ('}'). (Each type of component has its own properties; the properties for Darwin2K's current component types are listed in Table 2.1). The type and name of any component dependencies are then given in another list. If the component doesn't depend on any others, then this list is omitted. After defining a component, it can be included in a component list (as shown earlier in Figure 2.3), which is in turn included in a component context.

2.2 Connecting modules: the Configuration Graph

Parameterized modules describe only parts of a robot; an additional representation is needed to complete the robot description. The configuration graph is just that: it describes the way the modules are connected to each other. The configuration graph is a directed acyclic graph (DAG) in which nodes are modules and edges are physical connections between modules. The edges of the graph are *directed*: modules specify connections to their children via outgoing edges, but do not refer to their parent connections (incoming edges). The configuration graph is stored as a list of topologically sorted modules, and cycles in the graph are prevented by only allowing modules to specify attachments to modules which occur later in this list. Each module in a configuration can specify an attachment to another module for each connector. Attachments contain several fields: the ID of the connector on the parent module, the index of the child module, the ID of the con-

```
gearBox "maxon32.110374" {
  l = 0.05; d = 0.032; m = 0.194;  mt = 6.75;  ct = 4.5;
  r = 190;  e = 0.7;  v = 5000.0;
} {
  rotaryActuator "maxonRE35.118778";
  rotaryActuator "maxonRE36.118800";
}
```

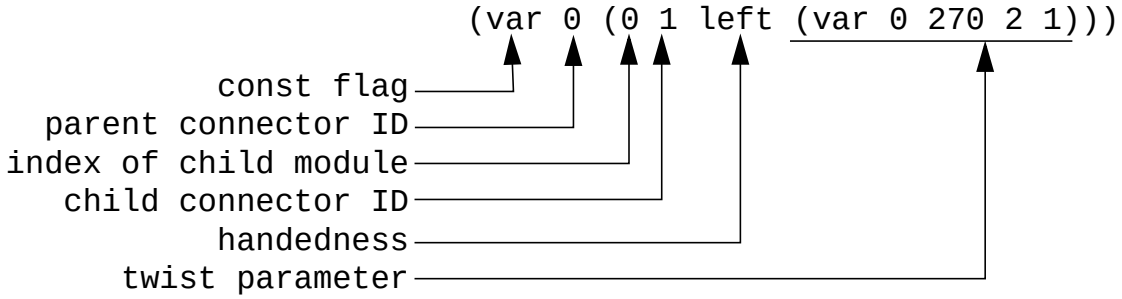
Figure 2.4: Component description for a gearbox

The first part of the component description gives values for the component's properties, such as gear ratio, length, diameter, mass, and maximum torque rating. The second part specifies the other components with which this component is compatible. In this case, the only motors in the componentDB file that the gearbox may be used with are the Maxon RE35.118778 and the Maxon RE36.118800.

Table 2.1: Component properties

component type	property	units
rotaryActuator	length	m
	diameter	m
	mass	kg
	stall torque	Nm
	continuous torque	Nm
	no load speed	rpm
	rotor inertia	$kg\ m^2$
	stall current	A
	resistance	Ω
	torque constant	Nm / A
	speed constant	rpm / V
	speed-torque gradient	rpm / Nm
gearBox	length	m
	diameter	m
	mass	kg
	maximum torque	Nm
	continuous torque	Nm
	gear ratio	none
	efficiency	none
	maximum velocity	rpm
material	density	kg / m^3
	modulus of elasticity	Pa
	shear modulus	Pa
leadScrew	mass per length	kg / m
	maximum force	m
	diameter	m
	efficiency	none
	rotations per distance	rad / m

nector on the child module, a `const`-flag, handedness, and a twist parameter:



The connector IDs simply identify which connectors are being attached; the twist parameter indicates the angle of rotation about the z-axis of the child's connector with respect to the parent's connector. The handedness can be either `left`, `right`, or `inherit`, and indicates whether the module's geometry should be normal (`left`) or a mirror image (`right`) A handedness of `inherit` means that the module should have the same handedness as its parent module. As with parameters, the `const`-flag can be used to indicate that a property (in this case, the module and connector references) should not be changed by the synthesizer. This allows the designer to specify that a particular sequence of modules should be untouched by genetic manipulations. For example, if the designer knew that a particular wrist configuration and end effector are required, then she would set the `const`-flag for the attachment between the wrist module and end effector. Thus, the designer can easily add significant constraints on the final form of the synthesis results, effectively incorporating task-specific knowledge and human expertise into the synthesis process. Figure 2.5 shows the text representation of a simple robot, and its physical instantiation.

For the purposes of simulation and analysis, the configuration graph is instantiated into a mechanism consisting of links (rigid bodies) connected by joints. This process is performed recursively, starting with the base and proceeding in a depth-first manner. Each module creates a physical description itself (including the shape of the module, the location of connectors, and any joints within the module), and then the physical representations for all modules are connected as specified by the configuration graph. Parts of different modules that are rigidly attached via connections are grouped into rigid bodies, whose inertial properties are then calculated.The `createGeometry` method is called for each module, and the `parts` on either side of inter-module connections are then attached to each other by a rigid connection. (All connections between modules are rigid; the only non-rigid connections are those forming joints between parts within the same module.) After all modules have created their geometric representation, the parts that are rigidly connected (as opposed to those connected by translating or rotation connections) are grouped into rigid bodies, and the inertial properties of each rigid body are computed using Coriolis [Baraff96]. The mechanism is thus represented by a tree, with nodes representing rigid bodies (links) and edges representing joints between bodies. The root of the tree is the one of the base's links. Figure 2.6 shows the configuration graph and mechanism (links and joints) representations for a simple, non-branching manipulator. After creating the mechanism representing the robot, the mechanism tree is traversed to identify serial chains. The mechanism and serial chain representations are used by numerous algorithms during the evaluation process, such as computing the robot's Jacobian or dy-

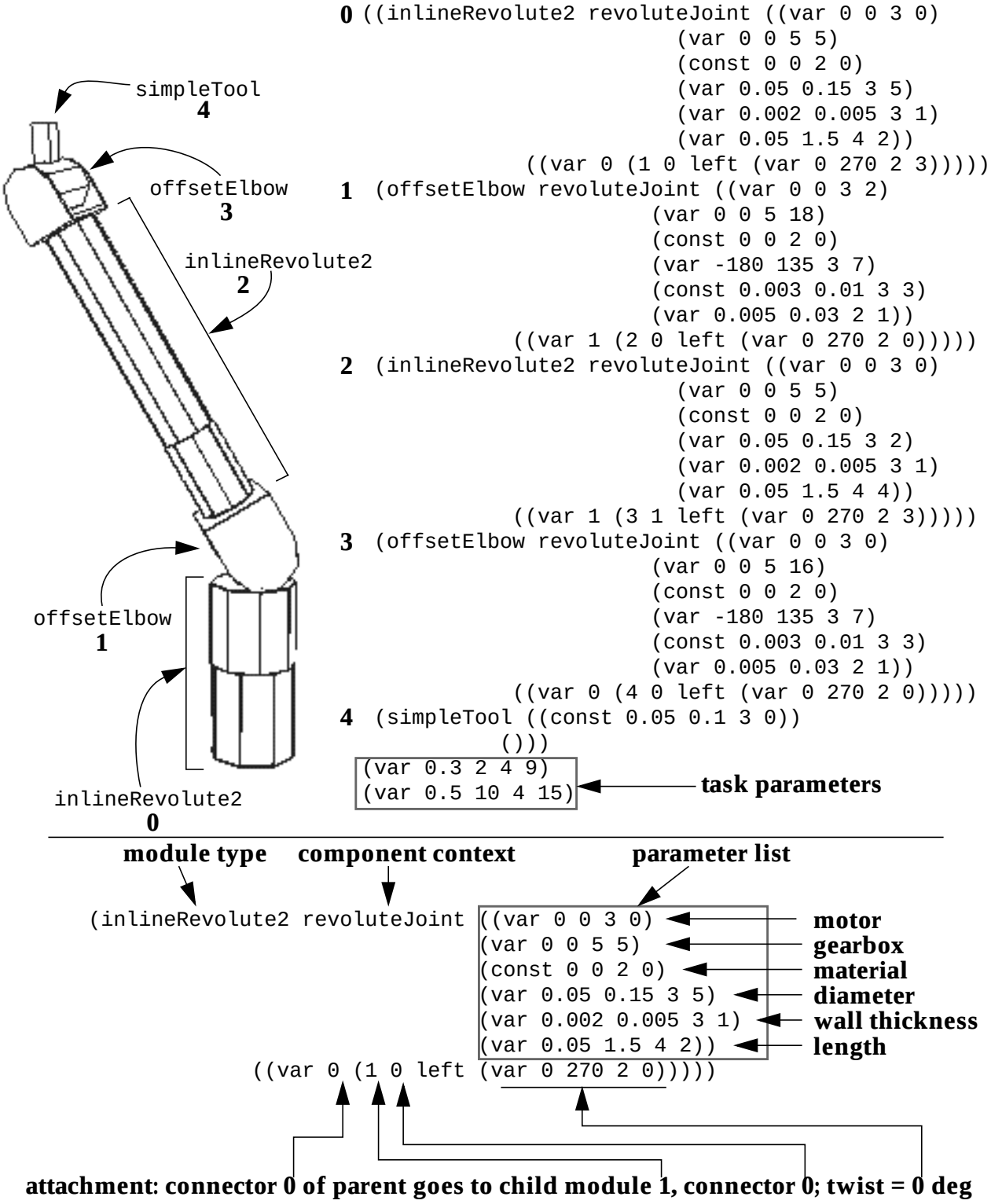


Figure 2.5: Sample configuration and text description

A 4-DOF manipulator constructed from five parameterized modules and with two task parameters is shown at top left, along with its text description (top right). Each module is labeled with a bold number to show the correspondence between the robot and text. At bottom is module **0**, with labels detailing parts of the module description.

namic model.

While the mechanism graph is a tree, the configuration graph may not be: when the graph is interpreted as mechanism, multiple connections referring to the same module will result in multiple copies of that module and its children. This allows pieces of a mechanism to be duplicated to preserve symmetry. For example, a hexapod robot would have one base with 6 connectors, each of which is attached to the same module (the base of a leg) in the configuration graph. Figure 2.7 shows an example of a free-flying robot with two identical arms. Note that there are two connections from the second module to the third; thus, the third module (and all its child modules) are duplicated, creating two identical arms. Creating parallel mechanisms in Darwin2K can be done through the use of constraints when dynamic simulation is used, or through manually-specified kinematic equations when kinematic simulation is used. The use of constraints will be discussed in a later chapter.

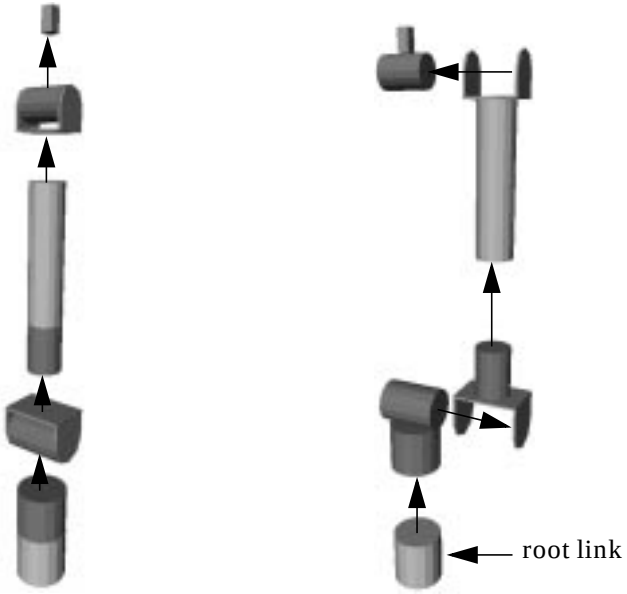


Figure 2.6: Configuration shown as modules and links

(a) shows the configuration from Figure 2.5 as a configuration graph, with modules connected by attachments (arrows).
(b) shows the robot represented as a mechanism, with links connected by joints. In this figure, the joint axes are represented by arrows, and the links have been displaced along the joint axes for clarity. The root link of the mechanism is indicated by the dashed arrow.

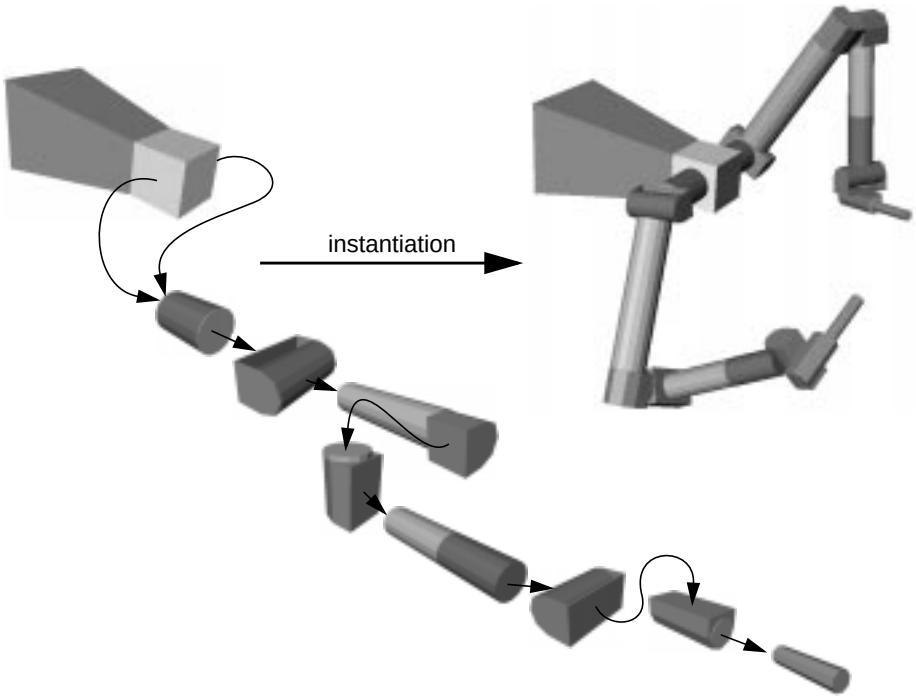


Figure 2.7: Configuration graph for a two-armed robot
At top is a symbolic view of a configuration graph. Each node is a module, and each edge is a connection between modules. The base module (at the left) has two outgoing connections to a joint module; when the graph is parsed to create a description of the robot's links and joints, the subgraph rooted at the joint module is duplicated, thus preserving symmetry.

2.3 `displayCfg`: Displaying and manipulating configurations

When specifing a configuration graph, it is useful to be able to view the resulting robot and interactively move its joints. The `displayCfg` program mentioned earlier is used for this. Figure 2.8 shows the GUI control panel for displayCfg. The chain selector and joint selector allow you to choose a joint (from the specified serial chain) to move with the joint slider. You can also enter the value for a joint angle (or distance, in the case of prismatic joints) with the joint angle input box. If you want to save a snapshot of the current window, you can click the `Save` button. This will write two files: `basename0.rgb` and `basename0.iv`, where basename is the name specified in the `Output basename` section of the GUI and `0` is the frame number, also specified in the GUI. Clicking `Save` increments the frame number, so that you don't have to enter a separate basename or frame number by hand each time you save an image. The `.rgb` file is written in the SGI

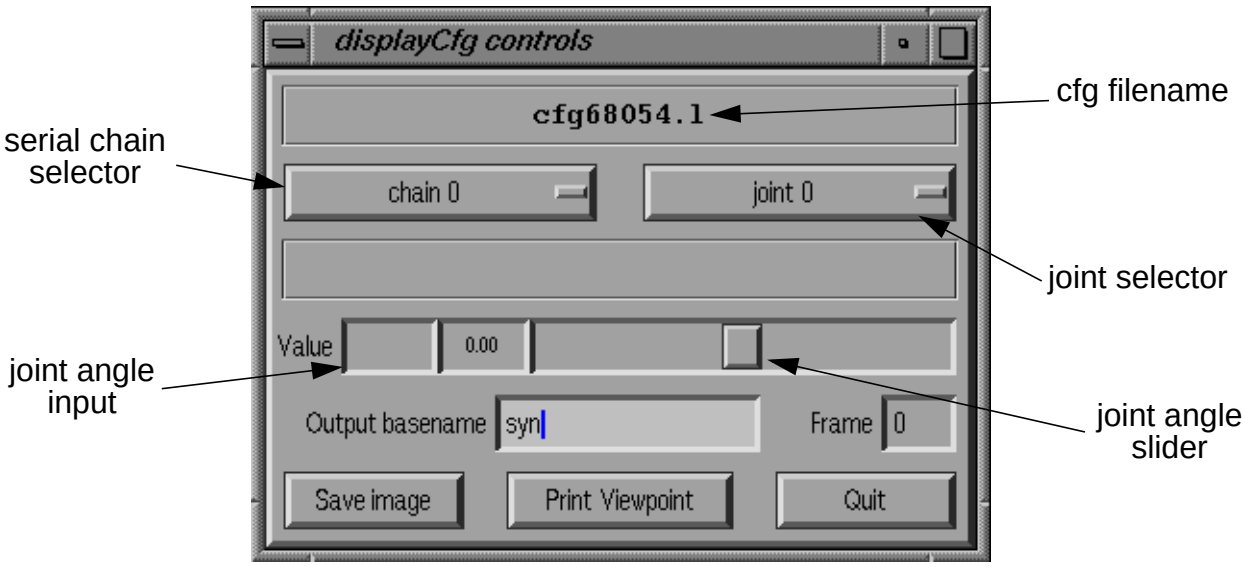


Figure 2.8: `displayCfg` **GUI**

file format, while the `.iv` file is an Inventor file that can be later viewed with `ivview` (on an SGI or other platform with Inventor installed) or with a web browser capable of viewing VRML files.

By default, `displayCfg` looks for a file called `disp.p` containing values for various display settings (e.g. whether to use a high- or low-detail display, or where the initial viewpoint should be) and for `componentDB` (the componentDB file). Alternate filenames for these files can be specified on the command line after the configuration filename. The componentDB file format was described earlier, but `disp.p` is written in another format called the *p-file* format. This format looks quite similar to the C language and is used throughout Darwin2K for specifying values for internal program variables.

Figure 2.9 shows a sample `disp.p` file consists of a number of variable definitions. The supported variable types are `int`, `float`, `double`, `char`, `char[]`, and `enum`,

```
int lowDetailDisplay = 1;
int displayWorldAxes = 0;
int displayFreeAxes = 0;
int displayPartAxes = 0;
int displayUsedAxes = 0;
int lowDetailTubeNumSides = 32;

#synIvDisplay
float backgroundR = 1.0;
float backgroundG = 1.0;
float backgroundB = 1.0;
```

Figure 2.9: `disp.p`

Shown here is disp.p, the p-file used by displayCfg. The p-file format is a list of variable assignments in a C-like syntax.

and they are specified in the same manner as in C. Each program looks for certain variables in its p-file; some variables are optional, while others are required. `displayCfg` has no required variables. The variables listed in Figure 2.9 affect the appearance of configurations in `displayCfg`: `lowDetailDisplay` determines whether a low- or high-detail display should be used; the next four variables indicate whether various coordinate axes should be shown; and `lowDetailTubeNumSides` specifies the number of facets to use when rendering parts with cylindrical geometry. The only line in the file that doesn't quite look like C is "`#synIvDisplay`", which is a context specifier: it means that the following variables should be grouped together and should only be readable by software objects asking for `synIvDisplay` variables specifically. Later, when we are building simulators, we will see that each software component (controller, payload model, etc.) has its own section in the p-file, allowing each component to be easily and consistently configured. In this case, the variables under `#synIvDisplay` are used by the software object that actually renders the robot, and the three variables shown specify the background color of the display. If you click on the `Print Viewpoint` button in `displayCfg`, you will see output that looks something like this:

```
int VPreadParams = 1;
int VPviewportMapping = 3;
int VPwidgetSizeX = 551;
int VPwidgetSizeY = 588;
float VPpositionX = -10.5446;
...
```

If you cut-and-paste this into `disp.p` under the `#synIvDisplay` section, then the next time you start `displayCfg` the viewpoint (including window size) will be restored to whatever the current viewpoint is.

2.4 Summary

In this chapter, we saw how to describe robots and view them with Darwin2K. Robots are composed of parameterized modules, and can be connected together to form configuration graphs. Module parameters can represent continuous properties like dimensions, or can select from a number of discrete choices such as components. Component properties are specified through the componentDB file, as are lists of allowable components for modules' selection parameters. We also saw the p-file format, used throughout Darwin2K for specifying interal program variables.

3 Using the Synthesizer

4 Simulation

5 Reference

This section describes the usage of Darwin2K's programs and software objects. For programs, the command-line arguments and p-file variables are detailed; for software objects, the behavior and interfaces are given. One might say that this is and "evolving" document, but at the present time "incomplete" is probably a more accurate description.

5.1 p-files

p-files contain user-specified runtime values for Darwin2K's programs. p-files look very much like lists of C variable definitions. For example, part of a p-file might look like this:

```
char evalType[80] = "walkerEvaluator";
char componentDBFilename[80] = "p/walker/walkerDB";

#evaluator
int numComponents = 6;
double maxRealtime = 1000.0;
```

In this case, most of the lines specify values of various types: the string `evalType` should have a value "`walkerEvaluator`", and so on. Different programs read different variables from p-files; these are detailed in the section for each program. When describing p-file variables, the following conventions are used:

- plain - the variable is optional and need not be specified
- **bold** - the variable is required
- *italic* - the variable is optional and is also conditional. It is read depending on the value of another variable, and need not be specified.
- *bold italic* - the variable is required and is conditional: when another variable has a certain value, this variable must be specified.

The other line in the p-file above is "`#evaluator`", which denotes the beginning of a new section or context. Typically, a p-file will contain information for a number of different software objects, and each object will have its own section. Depending on the program, the label for each section may be a classname or a predefined string plus a number number (for example `#metric2` or `#evComponent16`).

Note that when describing the p-file variable for each software object in this section, only those variables that are specifically introduced by a class are listed; the variables read by a class's base class are listed only under the base class.

5.2 Class and program reference

This section contains documentation on the following classes and programs:

```
pm - the population manager
evaluator - configuration evaluation program
evStandalone - standalone configuration evaluation program

synObject
     pmComponent
          cfgFilter
               dofFilter
               endPointFilter
               moduleRedundancyFilter
     evaluator
          pathEvaluator
          walkerEvaluator
          ffEvaluator
     evComponent
          collisionDetector
          genericPath
               path
                    relativePath
          payload
               panelSection
          DEsolver
               rungeKutta4
          controller
               sriController
                    ffController
               pidController
               roverController
```

pm - the population manager

The population manager `pm` is the center of Darwin2K's synthesis capabilities. It creates configurations, sends them to evaluator processes for performance assessment, and uses the performance data to select above-average configurations to be reproduced.

Usage:

```
pm p-file
```

General setup variables

Variable	Description	Default value
int readPopulation	when 1, the pm will read the initial population rather than generating one.	-
char popFilename[]	if readPopulation is 1, this variable specifies the filename containing the initial population	-
int readFitness	if readPopulation is 1, this variable specifies whether fitness information for each configuration is stored in the population file.	1
int reevaluatePopulation	if readPopulation is 1, this variable determines whether or not the configurations in the initial population will be re-evaluated (i.e. sent to evaluators for fitness evaluations)	0
int saveAndQuitAfterCreation	when 1, the pm will generate an initial population, save it to the file 'initPop.l', and exit without performing any optimization.	0
int saveAfterDecimation	when 1, the pm will save the population to a file 'init.l' after decimation takes place.	1
int creationTimeout	the number of configurations the pm will create while attempting to generate the initial population before giving up.	10*initPopSize
int debugLimit	when positive, forces the pm to exit the optimization loop after the indicated number of configurations have been evaluated.	-1
int minLimit	when using Requirement Prioritization, this variable specifies the minimum number of configurations that are evaluated after generating the first configuration that is feasible according to the current requirement group.	max(2000, 20*popSize)

Variable	Description	Default value
int fillAtStartOfGroup	when using Requirement Prioritization, this variable indicates whether or not the population should be filled with randomly-generated configurations after advancing to the next requirement group.	1
int preventDuplicates	when 1, indicates that duplicate configurations (exact copies of other configurations) should not be allowed into the population.	1
int popSize	the number of configurations in the population during the optimization phase	100
int initPopSize	the number of configurations initially generated. After evaluating the initial population, configurations are stochastically selected for deletion until the population is of the size indicated by popSize.	1000
int maxIndividuals	When using requirement prioritization, the pm will continue generating configurations until the indicated number of evaluations have occured since advancing to the current group of requirements. When not using requirement prioritization, or when stopAtMax is 1, the pm will always stop after the specified number of evaluations have been performed.	60000
int stageLimit	When using requirement prioritization, the pm will continue to evaluate configurations until the indicated number of evaluations have been performed since advancing to the current requirement group, or since satisfying the current requirement group.	20000
int stopAtMax	Forces the pm to stop after the number of evaluations indicated by maxIndividuals	0
int numTaskParams	The number of task parameters included in each configuration.	-
char evalType[]	The class name of the evaluator to be used for evaluating configurations.	-
char evalParamFile[]	The p-file for the evaluator. This filename should be either an absolute filename or should be relative to the user's login directory, since only the filename (rather than the file itself) is sent to evaluator processes.	-
char componentDBFilename[]	The componentDB file. This filename should be either asbolute or relative to the user's login directory.	-
char moduleDBFilename[]	The module database file. This filename should be either absolute or relative to the user's login directory.	
char kernelFilename[] (or embryoFilename)	The kernel file. This filename should be either absolute or relative to the user's login directory.	

Logging and data collection

Variable	Description	Default value
int logPopulation	indicates whether the entire population should be written (to pop.l)	1
int logAllOptimalSets	indicates whether optimal sets should be written successive files (e.g. optimalNNNN.l) or should always be written to optimal.l	0
int optimalLogInterval	when logAllOptimalSets, this varaiable indicates the number of evaluations between saving the optimal set to disk.	100
int logInterval	when logPopulation is 1, this variable determines whether the population is always written to pop.l (logInterval <= 0) or to successive files (e.g. popNNNN.l, where NNNN is a multiple of logInterval).	-1

Genetic Operators

Variable	Description	Default value
int doSubgraphFixing	When commonality-preserving crossover is being used (i.e. cpXoverRate > 0), this variable determines whether or not subgraph fixing should be performed.	1
int useAdaptiveMutation	Indicates whether adaptive mutation should be used.	0
double adaptiveMutationTimeConstant	When adaptive mutation is used, this number multiplies the population size to determine the time constant of the exponential function determining total mutation probability.	5
int reuseEmbryos	When fillAtStartOfGroup is 1, this variable determines whether or not the embryos should be randomly chosen as parents for new configurations.	0
double cpXoverRate	Commonality-preserving crossover rate. The three crossover rates should sum to a number between 0 and 1, and the duplication rate is one minus this sum.	0.1
double xoverRate	Subgraph crossover rate.	0.7
double paramXoverRate	Parameter crossover rate.	0.15
double paramMutationRate	Parameter mutation rate.	0.02
double attMutationRate	Attachment mutation rate.	0.01
double replacementRate	Um....module replacement rate?	0.01

Variable	Description	Default value
double insertionRate	module insertion rate	0.01
double deletionRate	module deletion rate	0.01
double permuationRate	module permuation rate	0.01

Metrics and selection

Variable	Description	Default value
int numMetrics	The number of metrics	-
enum selectionMode	One of {weightedSum, cdf, mdf, req }. req is the suggested value, though cdf is also useful. The other modes are not recommended.	req
int newMetricSpec	Indicates whether the new or old metric specification format should be used. The old format is undocumented and is only used for backwards compatibility.	1
int reqStartGroup	when selectionMode is 'req', the pm automatically advances to the indicated requirement group. This is useful when continuing a previous run.	0
char cdfFilename[]	when selectionMode is 'cdf', this variable is used to determine the name of the file containing the CDF definition.	-
char fdfFilename[]	when selectionMode is 'cdf', this variable is used to determine the name of the file containing theFDF definition.	-

The following variables must be specified for each metric, and are contained in sections named '#metric<num>' where <num> is a number from 0 to numMetrics-1.

Variable	Description	Default value
char name[]	The class name of the metric	-
enum mode	For state-dependent metrics, this specifies the method used to collapse the time-varying performance measurements into a single metric. Legal values are MIN, MAX, AVG, RMS (root-mean-square), and INTEGRAL.	-
double min	Specifies the lower-bound to which raw metric values are clipped.	-
double max	Specifies the upper-bound to which raw metric values are clipped.	-

Variable	Description	Default value
double afdi	The Adjusted Fitness Doubling Increment for the metric. A benefical change in raw fitness of the specified amount will result in a doubling of adjusted fitness. Either this variable or scale (below) must be specified.	-
double scale	The scale factor that is used to compute adjusted fitness from standardized fitness as follows: $a = e^{scale \times s}$ where *s* is the standardized fitness and *a* is the adjusted fitness. Either this variable or afdi (above) must be specified.	ln(2) / afdi
int priority	When requirement prioritization is being used, this variable specifies the priority level of the metric.	-
char op[]	When requirement prioritization is being used, this variable specifies the comparison operator to use when testing feasibility. Valid values are: ==, !=, <, <=, >=, >, and NOP. The equality and inequality operators behave as in C, while NOP indicates that there is no feasibility threshold.	-
double thresh	When requirement prioritization is used and op is not equal to NOP, this variable specifies the acceptance threshold for the metric.	-

Source file

`pm/*`

See also

`pmComponent`

synObject

`synObject` is the abstract base class for all classes in Darwin2K that use runtime typing. This allows objects of any class derived from synObject to be created at runtime through Darwin2K's synDB, a database providing a mapping from class names (represented as character strings) to constructors. While there is some overhead involved in defining classes so that they will be properly accessed by the synDB, new classes can be added through dynamic libraries and instantiated on the fly without recompiling Darwin2K's binaries.

The synDB capabilities make heavy use of macros, so that users defining new classes can easily add the appropriate magic to get their classes working correctly. The primary macros needed when writing a new class are:

`MAKE_COVER_FUNCTIONS(classname)` - declares the magic functions for classes without explicity constructors or with constructors that have no arguments

`MAKE_ABSTRACT_COVER_FUNCTIONS(classname)` - declares magic functions for classes that are pure virtual

`MAKE_COVER_FUNCTIONS_NO_CONS(classname)` - declares magic functions for classes whose constructors require arguments

These macros should be put in the class definition; for example

```
class exampleObj : public synObject {
public:
  MAKE_COVER_FUNCTIONS(exampleObj);
  DECLARE_PARENT_CLASS(synObject);
};
```

In this case, the new class does not have a constructor so the normal `MAKE_COVER_FUNCTIONS` macro is used. The `DECLARE_PARENT_CLASS` macro is also included, with the name of the parent class as the argument. This allows the full inheritance of the object to be known at runtime. For classes with multiple inheritance, the macros

```
DECLARE_PARENT_CLASSES(class1, class2)
DECLARE_PARENT_CLASSES3(class1, class2, class3)
```

can be used. In addition to the macros described above, the source file (as opposed to header file) for any new class should include the line

```
DEFINE_CLASS_ID(classname)
```

This defines an integer that is a static member of the class, and which contains a unique ID number for the class. Finally, the macro

```
ADD_DB_TYPE(baseClass, classname, initFunc)
```

adds the class's `copy()` method to the synDB database. `initFunc` is a normal function or a static member function, and if it is non-NULL then the database will call `initFunc` once at program startup. Any class-specific initialization that needs to be performed should thus be done by `initFunc`. This macro is normally used in `initializeUserEvalDB` and `initializeUserPMDB`, functions which application-specific dynamic libraries must define.

When the above macros are used, runtime typing can be performed using the macro

```
IS_OF_TYPE(obj, classname, derivedOK)
```

which returns 1 if `obj` is of the specified class, or if `derivedOK` is 1 and obj is derived from the given class or is a member of it. All of these macros are defined in `db/synMacros.h`.

Data memebers

`int objectID` - a general-purpose ID that derived classes can use
`int verboseLevel` - indicates the desired level of text output from the object

```
enum {
  SO_NONE = -1,‘      / / don’t print anything
  SO_ERRORS = 0,      / / only print errors
  SO_WARNINGS = 1,    / / errors, warnings
  SO_INFO = 2,        / / errors, warnings, occasional useful data
  SO_DEBUG = 3,       / / full output; lots of data spewed (every sim step)
  SO_ULTRASPEW = 4    / / hope you have lots of free disk space
};
```

New methods

`virtual ~synObject(void)` - a virtual destructor
`virtual const char *className(void) const` - (pure virtual) Returns a character string containing the object's class name
`virtual synObject *copy(void)` - (pure virtual) Returns a new object that is a copy of the current object. Thus, even if the type of an object derived from synObject is unknown, one can always create a copy of it.
`virtual int isOfType(int typeNum, int derivedOk)` - returns 1 if the object is of the specified type, or derived from the specified type if derivedOk is 1.

Source file

```
db/synObject.h
db/synMacros.h
```

pmComponent

`pmComponent` is the abstract base class for modular software components used by the population manager. `pmComponents` read their initialization data from the pm's p-file. Currently there is only one class derived from `pmComponent`, the `cfgFilter`.

Derivation

```
pmComponent : synObject
```

New methods

```
int readParams(paramParser *p)
```

p-file variables

Variable	Description	Default value
int enabled	Determines whether or not the pmComponent is used. Note that this is actually read by the pm before creating the pmComponent.	1
int verboseLevel	Sets the level of text output.	1 (SO_WARNINGS)

Source file

```
db/pmComponent.h
```

See also

```
pm - the population manager
cfgFilter
```

cfgFilter

The `cfgFilter` is an abstract base class implementing a generic filter used by the pm to remove configurations that do not meet user-defined constraints. These constraints are inherent properties of the configuration (e.g. number of degrees of freedom), as opposed to performance measurements that must be obtained through simulation.

Derivation

```
cfgFilter : pmComponent : synObject
```

New methods

`int acceptable(configuration *cfg)`- returns 1 if the configuration passes the filter, 0 if not.

Source file

```
pm/cfgFilter.*
```

See also

```
pm - the population manager
pmComponent
dofFilter
endPointFilter
moduleRedundancyFilter
```

dofFilter

The `dofFilter` is a `cfgFilter` that filters based on the number of degrees of freedom. Configurations must have at least `min` degrees of freedom and at most `max`. The number of degrees-of-freedom is calculated by using the numDOFs() method of each module, rather than instantiating the configuration and counting the actual number of joints. This computes the number of *effective* degrees of freedom rather than the number of actual joints, which is important when modules have kinematic or dynamic constraints, or what a module has several completely redundant degrees of freedom (e.g. a prismatic joint with several telescoping sections). Also note that the 6 DOFs for free-flying bases (including the `virtualBase`) are counted by the dofFilter.

Derivation

```
dofFilter : cfgFilter : pmComponent : synObject
```

New methods

None.

p-file variables

Variable	Description	Default value
int min	The minimum number of degrees of freedom for acceptable configurations.	-
int max	The minimum number of degrees of freedom for acceptable configurations.	-

Source file

```
pm/cfgFilter.*
```

See also

```
pm - the population manager
cfgFilter
```

endPointFilter

The `endPointFilter` filters based on the number of end effectors. Robots must have at least `min` and at most `max` end effectors to be considered acceptable by this filter.

Derivation

```
endPointFilter : cfgFilter : pmComponent : synObject
```

New methods

None.

p-file variables

Variable	Description	Default value
int min	The minimum number of end effectors for acceptable configurations.	-
int max	The minimum number of end effectors for acceptable configurations.	-

Source file

```
pm/cfgFilter.*
```

See also

```
pm - the population manager
cfgFilter
```

moduleRedundancyFilter

The moduleRedundancyFilter is a cfgFilter that filters based on adjacent modules in the configuration graph. Its original purpose was to eliminate configurations that have two sequential degrees of freedom that are kinematically identical, such as two inlineRevolute joint modules connected in series. More generally, the moduleRedundancyFilter allows the designer to disallow connections between certain module types.

Derivation

```
moduleRedundancyFilter : cfgFilter : pmComponent : synObject
```

New methods

```
int pairAcceptable(int moduleID1, int connID1,
                   int moduleID2, int connID2) const
```

p-file variables

Variable	Description	Default value
int numIllegalPairs	The number of disallowed pairs of modules	-
*char moduleName1_*<**i**>[]	The class name of the first module in the i^{th} pair of illegal modules.	-
*char moduleName2_*<**i**>[]	The class name of the second module	-
*int connectorID1_*<i>	The connector ID for the connector on the first module that should not be attached to the second module. A connector ID of -1 indicates that no connection is allowed.	-1
*int connectorID2_*<i>	The connectorID for the connector on the second module.	-1

Source file

```
pm/cfgFilter.*
```

See also

```
pm - the population manager
cfgFilter
```

evaluator - configuration evaluation program

evStandalone - standalone configuration evaluation program

evaluator

At the highest level, the `evaluator` performs initialization for simulation components (called `evComponents`) and provides high-level simulation control. The `evaluator` class is a base class for other evaluators, and does not include any type of evaluation itself -- other classes such as the `pathEvaluator` can be derived from it to meet the needs of specific task representations. The `evaluator` defines the interface between the simulation internals and the rest of Darwin2K's software, and also keeps track of the configuration being evaluated and the metrics and `evComponents` being used for simulation. The evaluator also parses the initialization file to read its own parameters and to determine which `evComponents` are required, and then passes component-specific information from the file to each `evComponent`. If any task parameters are included in the optimization process, the evaluator passes each task parameter to its corresponding `evComponent` (or interprets the task parameter itself, if one of the `evaluator's` variables was specified as a task parameter). The `evaluator` also calls the appropriate initialization and cleanup functions for each configuration being evaluated and performs other bookkeeping functions. In addition to these functions, classes derived from the evaluator also contain code for controlling the simulation: for example, at each time step in a simulation the `pathEvaluator` tells a `DEsolver` to query a `controller` for a command and then compute the robot's state at the next time step, then checks if any collisions have occurred using the `collisionDetector`, and determines if the robot has completed the task by reaching the end of the current `path`. In many cases, the designer will need to derive a task-specific class from `evaluator` or one of it subclasses. The methods listed under "Core methods" below are normally provided by derived classes; other virtual functions (such as the display functions) can be overridden, but the default methods for the evaluator class will normally be suitable for derived classes as well. Other functions are convenience functions that make accessing data members easier (such as `getEvComponent()`) or that are useful for derived classes (e.g. `checkTimeout()`).

Normally, the program will create an evaluator (either directly, or through the `createEvaluatorFromFile()` static method) and initialization will be performed in a standard manner. Thus, derived classes should expect their methods to be called in the following order:

`readParams(p-file)` - reads class-specific parameters from a p-file.
`postComponentInit()` - called after all evComponents have been initialized

and for each configuration that is evaluated, the following methods will be called:

`init(cfg)` - performs any configuration-specific initialization such as memory allocation
`setVariables(taskParams)` - reads the values of any task parameters from the supplied list of parameters. Task parameters override values from the p-file.
`evaluateConfiguration()` - evaluates the configuration; this is the main simulation loop

`cleanup()` - frees any dynamic memory associated with the configuration.

NOTE: The methods `readParams`, `postComponentInit`, `init`, `setVariables`, and `cleanup` *must* call the base class's corresponding method, and each of these methods should fail (i.e. return 0) if the base class's method fails. `init` and `postComponentInit`() should call the corresponding base class method *before* doing any class-specific initialization; `cleanup` should call the corresponding method *after* class-specific operations; and the order for the other methods is unimportant.

The p-file for the `evaluator` and `evComponents` is usually parsed by `createEvaluatorFromFile`. This static method creates an `evaluator` of the type specified in the p-file, creates `evComponents` also as specified in the file, and calls all `evaluator` and `evComponent` initialization methods that are not specific to the configuration being evaluated (i.e. `readParams` is called, but `init` is not). Several conventions should be followed in the p-file:

- The variables specific to each evaluator class should be defined in a parser context with a label of #`classname`, where classname is the name of the evaluator class. For example, a `pathEvaluator` would actually require two sections in the file, one starting with #`evaluator` and containing `evaluator`-specific variables and one starting with #`pathEvaluator`.
- The p-file variable `numComponents` (int the `evaluator` context) indicates how many `evComponents` are defined in the file. Each `evComponent`'s variables are stored in a separate section, with the N^{th} `evComponents` variables in the section labeled #`evComponent`*N*. The classname for each component is specified by a string variable called `class`, and an optional integer variable `enabled` determines whether or not the `evComponent` is created and initialized.
- The string variable evalType is read directly by createEvaluatorFromFile and is thus part of the global context (i.e. it comes before the first context label).

Derivation

`evaluator : synObject`

Data members

`configuration *cfg` - the configuration currently being evaluated
`ptrList evComponents` - a list of evComponents.
`ptrList metrics` - a list of metrics
`double dt` - simulation stepsize in seconds
`double maxRealtime` - maximum elapsed time (wall clock time), in seconds
`int startPaused` - 1 if the simulation should initially be paused
`double simTimeout` - maximum elapsed simulation time, in seconds
`ulong startSecs` - wall-clock simulation start time, in seconds

`double currentTime` - current elapsed simulated time, in seconds

New methods

Core methods - usually provided by subclasses

`virtual int readParams(const char *fname)` - reads class-specific parameters from the given p-file. Returns 1 on success, 0 on failure. This function should call the base class's readParams() method, and should return 0 if the base class's readParams() fails. This method should call `EVparser.pushContext(`*classname*`)` before parsing class-specific variables, and should call `EVparser.popContext()` after parsing class-specific variables.

`virtual int init(configuration *cfg)` - allocates memory and sets initial configuration state. This method should *first* call the base class's init() method, and should return 0.

`virtual int cleanup(void)` - called after evaluateConfiguration to free memory allocated in init()

`virtual int setVariables(const ptrList *taskParams)` - setVariables is called after `init(cfg)`

`virtual int evaluateConfiguration(void)` - evaluates the configuration set by `init()`

`virtual int postComponentInit(void)` - called at the end of `initializeComponents()`

Convenience functions

`evComponent *getEvComponent(int i)` - returns the i^{th} evComponent.

evComponent *getEvComponentByLabel(const char *label) - returns the first evComponent that has the given label.

```
evComponent *getEvComponentByClassID(int classID, int
                                     derivedOk = 1)
```

returns the first `evComponent` with the given classID or derived from the given classID if `derivedOk` is 1. Normally used through the `getEvComponentsByClassName` macro.

```
int getEvComponentsByClassID(int classID, ptrList *l,
                             int derivedOk = 1)
```

adds all `evComponents` with the given classID (or derived from the given classID if `derivedOk` is 1) to l, and returns the number of appropriate `evComponents` found. Normally used through the `getEvComponentsByClassName` macro.

`metric *getMetric(int i)` - returns the i^{th} metric.

`metric *getMetricByName(const char *className)` - returns the first metric found that has the given classname.

`taskParamRecord *getTaskParam(int i)` - returns the i^{th} task parameter record.

`int numTaskParams(void)` - returns the number of task parameters records.

Macros

`#define getEvComponentByClassName(classname, derivedOk)` - returns the first evComponent found that has the given classname, or that is derived from the specified class if `derivedOk` is 1.

#define getEvComponentsByClassName(classname, l, derivedOk) - adds all evComponents

#define GET_METRIC_BY_NAME(name) ((name *)getMetricByName(#name));

Display methods

`virtual int usingDisplay(void)` - returns 1 if a display is being used

`virtual int parseDisplayVariables(paramParser *p)` - parses display-related variables.

`virtual void setPauseState(int paused)` - pauses and unpauses the simulation

`virtual void initDisplay(void)` - initializes display, including opening windows

`virtual int updateDisplay(void)` - updates display

`virtual void cleanupDisplay(void)` - closes display and frees any associated memory

GUImethods

`virtual void initGUI(void)` - initializes separate GUI window, if any

`virtual int updateGUI(void)` - updates GUI state, including checking for and handling any window manager events

`virtual void closeGUI(void)` - closes the GUI and frees any associated memory

Other methods

`static evaluator *createEvaluatorFromFile(const char *fname)`- creates an evaluator and evComponents from the descriptions in the given p-file and performs all evaluator and evComponent initialization.

`void resetTimer(void)` - resets the timer measuring elapsed CPU time

`int checkTimeout(void)` - returns 1 if the elapsed CPU time for the current evaluation is greater than simTimeout.

`void printMetrics(FILE *fp)`- prints the names and values of all metrics.

p-file variables

Variable	Description	Default value
int verboseLevel	Sets the level of text output.	1 (SO_WARNINGS)
int numComponents	Indicates the number of evComponents to be described in the file	-

Variable	Description	Default value
#evComponent<**N**>	This is a parser context rather than a variable, and denotes the beginning of the p-file section for the N^{th} evComponent.	
int numTaskParams	The number of task parameters to be included with each configuration.	0
char taskParamVariableName<N>[]	The variable name for the N^{th} task parameter, as expected by the evaluator or evComponent's setVariable() method.	-
char taskParamComponentLabel<N>[]	The label of the evComponent which the N^{th} task parameter is for. Omit this variable if the task parameter is for the evaluator.	-
double maxRealtime	maximum elapsed time (wall clock time), in seconds	30
int startPaused	1 if the simulation should initially be paused	0
double dt	simulation stepsize in seconds	
double simTimeout	maximum elapsed simulation time, in seconds	
int preventInitialPose	When 1, indicates that any initial joint values specified by modules should be ignored. Provided for backwards compatibility. Defined in cfg / configuration.cxx.	
int lowDetailTubeNumSides	The number of facets used for low-detail representations of cylinders. Defined in modules / componentJoints.cxx	
int constraintVerboseLevel	The verbose level for constrained dynamic simulation code. Defined in cfg / constraint.cxx	1 (SO_WARNINGS)
int patchVerboseLevel	The verbose level for terrain contact patch code. Defined in terrain / patch.cxx	1 (SO_WARNINGS)
int patchAverageNormals	See terrain / patch.cxx	
int intersectVerboseLevel	The verbose level for polyhedron intersection calculation code. See collision / intersect.cxx	1 (SO_WARNINGS)
int cfgVerboseLevel	Verbose level for general configuration-related code. See cfg / configuration.cxx	1 (SO_WARNINGS)
int cfgDynamicVerboseLevel	Verbose level for dynamic simulation code. See cfg / dynamics.cxx	1 (SO_WARNINGS)

Variable	Description	Default value
int bombOnError	When 1, any error (i.e. a call to logError()) will cause a core-dump so that post-mortem debugging can be performed.	0
int randomSeed	When specified, this is used as the argment to srand48(). When not specified, srand48() is not called.	-

Source file

```
eval/evaluator*
```

See also

```
pm - the population manager
evComponent
```

pathEvaluator

The `pathEvaluator` represents tasks as a series of end-effector trajectories, optionally with obstacles in the workspace and payloads to be moved along the trajectories. In the simplest case, a fixed-base manipulator might follow a single trajectory; a more complex task might require a mobile robot with multiple end-effectors to move between base poses and follow paths with one or both manipulators at each pose. To account for this variation, the trajectories (represented as `path` or `relativePath` objects) are organized in *path groups*: each path group can contain a trajectory for each of the robot's end-effectors, and can have a different pose for the robot's base. If multiple paths are specified in a path group, the corresponding manipulators follow them simultaneously and all paths must be completed before moving on to the next path group. The `pathEvaluator` requires several `evComponents` to be specified in the initialization files: a `DEsolver` for integrating robot state, an `sriController` for controlling the robot, and one or more `paths`. Additionally, the `pathEvaluator` can use a `motionPlanner` to plan paths between path groups for the robot's base, and a `collisionDetector` to check for collisions during simulation.

The initial position and orientation of the robot's base can be specified in the initialization file, or can be included as task parameters so that the base position of a manipulator can be optimized. However, when using a mobile base a different base location may be required for each path group, and different mobile robots may require different base poses in order to reach the same path since they may have different manipulator kinematics. The `pathEvaluator` uses the `sriController` in a two-stage process to determine the base pose of mobile robots for each path group: at first, the robot's base is allowed to move freely (i.e. without nonholonomic constraints), and then all of the robot's degrees of freedom (including the base) are moved. If we were to initially use all of the robot's degrees of freedom, then the serial chains might end up at the edge of their workspaces. Instead, the `pathEvaluator` first allows the base to move freely in 3 dimensions (or in the plane for a planar base) while holding all serial chains fixed. Once the robot cannot move any closer to the start of the paths in the path group, serial chain motions are enabled but with a high cost (implemented by scaling columns in the Jacobian corresponding to serial-chain DOFs), forcing the robot to move the base instead of the serial chains if possible. During this second stage, the robot base is still able to move freely. If the robot still cannot reach the initial points in the paths, we know it cannot complete the task. If it does find a successful starting pose, the base motion mode is returned to normal (i.e. non-holonomic constraints are enforced, for example) and the serial chain cost is returned to normal; this is the normal operating mode for the robot.

If the task requires the robot to move between several base poses, the aforementioned method is used to compute each initial base pose before the actual evaluation takes place. During evaluation, the `sriController` can be used to move holonomic robots between base poses if there are no obstacles present. When a robot with a nonholonomic base moves between base poses, or if there are obstacles in the workspace, the robot can be controlled using the `motionPlanner`; however, note that the `motionPlanner` class has not been used in several years and likely suffers from bit-rot.

Derivation

```
pathEvaluator : evaluator : synObject
```

Data members

`evComponents` used by the `pathEvaluator`. Listed with each variable is the label or class used to select the appropriate evComponent from those defined in the p-file.

`collisionDetector *cd` - the collision detector. Labeled "`collisionDetector`", or of type of `collisionDetector`. (Optional)

`motionPlanner *mp` - the motion planner; of type `motionPlanner`. (Optional)

`sriController *ctrl` - the controller to use for trajectory following; of type sriController

`DEsolver *dynSolver` - the DE solver used during dynamic simulation; labeled "dynamicSolver". Either `dynamicSolver` or `kinematicSolver` must be specified in the p-file

`DEsolver *kinSolver` - the DE solver used during kinematic simulation; labeled "`kinematicSolver`".

`DEsolver *mpSolver` - the DE solver used during planned motions; labeled "`plannerSolver`". (Optional; `kinematicSolver` is used if `plannerSolver` is not defined.)

`ptrList paths` - a list of path groups; each path group is a ptrList of paths for the robot's end effectors.

variables describing path groups and behavior

`int numBasePoses` - number of path groups (each of which may have a different base pose for the robot, if the robot has a mobile base)

`int numEndPoints` - maximum number of paths in each group

`int doBaseMoves` - 1 if the motions between path groups should be simulated

`int baseFixed` - 1 if the base is fixed. 0 indicates that base poses for each path group should be computed.

`int useDynamicSimulation` - indicates whether dynamic or kinematic simulation should be used.

`double baseCost` - cost of base motions during normal movements

`double simChainCost` - cost of serial chain motions during normal movements

`double chainCost` - cost of serial chain motions while finding the base pose for each path group

`double posThresh[3]` - the linear velocity thresholds (in m/s) used for deciding when the robot has stopped moving during each of the three phases.

`double orientationThresh[3]` - angular velocity thresholds (in rad/s) for deciding when the robot has stopped moving during each of the three phases.

`triple origin` - the initial position of the origin of the robot's base link

`quaternion originOrientation` - the initial orientation of the robot's base link

`vector **startState` - array of vectors containing the computed starting pose for each path group

`double **basePose` - the pose of the base for each path group; used by the motionPlanner

New methods

`int findBasePoses(void)` - if a robot has a mobile base, this method computes a starting pose for each path group by first moving on the robot's base, then moving the robot's chains (though with a high cost) until the robot's end effectors are at the start of their respective paths.

`genericPath *getPath(int whichPose, int i)` - returns the i^{th} path for the path group indicated by `whichPose`.

`int numPaths(int whichPose)` - returns the number of paths in the indicated path group.

p-file variables

Variable	Description	Default value
int useDynamicSimulation	See 'Data members' above	-
int numEndPoints	See 'Data members' above	-
int numBasePoses	See 'Data members' above	-
int doBaseMoves	See 'Data members' above	-
int baseFixed	See 'Data members' above	0
double baseCost	See 'Data members' above	1.0
double simChainCost	See 'Data members' above	1.0
double chainCost	See 'Data members' above	1.0
double posThresh{0,1,2}	See 'Data members' above	0.1, 0.001, 0.001
double orientationThresh{0,1,2}	See 'Data members' above	1, 0.1, 0.01 * π / 180
double originPos{X,Y,Z}	See 'Data members' above	{0, 0, 0}
double originOrientation{R,I,J,K}	See 'Data members' above	{1, 0, 0, 0}

Task parameters

Variable	Description
double originPos{X,Y,Z}	See 'Data members' above

Variable	Description
double originOrientation{R,I,J,K}	See ‘Data members’ above
double originHeading	Rotation about z axis (up) for origin; used in place of originOrientation.

Source file

```
eval/pathEvaluator.*
```

See also

```
evaluator
evComponent
```

walkerEvaluator

The `walkerEvaluator` simulates a robot moving along a truss in a zero-gravity environment. The task involves walking along one segment (rod) of a truss, moving to a perpendicular rod, moving to a third rod, then rotating about the rod and positioning one end effector beneath a truss junction for inspection purposes. The class is derived from the `pathEvaluator`, as it shares many of the same simulation needs; however, the `pathEvaluator`'s path groups and base-pose computation code are not used. Instead, the `walkerEvaluator` uses several `evComponents` and specialized classes to generate trajectories for the robot. The `panelSection` represents a triangular solar panel backed by a truss, which the robot walks along. The `trussPath` class computes a series of trajectories (each stored as a `relativePath`) that represent a hand-over-hand gait from one end of a truss rod to the other. The `trussPath` has several parameters determining the geometry of the gait; these parameters are set by the `walkerEvaluator`, which in turn obtains their values from the p-file and from a configuration's task parameters. The `walkerEvaluator` (rather than the `trussPath`) computes the transition paths between truss segments, as well as the final inspection path.

Derivation

```
walkerEvaluator : pathEvaluator : evaluator : synObject
```

Data members

`double pathTimeout` - maximum simulation time allowed for each path

`double walkingWeight` - the relative importance of the walking phase of the task when computing task completion.

`double transitionWeight` - the relative importance of *each phase* of the truss transition portion of the task when computing task completion.

`double transitionDist` - the distance from the end of the rod for the final grasp point along a segment.

`double inspectionStandoff` - the distance from the truss junction to the TCP during the inspection phase.

`double stride, hOffset, vOffset, approachAngle, approachDist, viaTol, tol, vTol, angleTol` - gait parameters; see Figure 5.40 in section 5.6 of *Darwin2K: An Evolutionary Approach to Automated Design for Robotics* for descriptions of these.

`int firstGraspPoint` - determines which gripper the robot initially uses to grab the truss.

`int phase` - the current phase of the gait; 0 indicates that the distal link is moving, and 1 indicates that the base link is moving.

`int stopAfterWalking` - determines whether or not the task includes the transition phase or only the walking phase.

`int computeLinkPath` - determines whether the relativePath's `computeLinkPath` variable is 1 or 0 when moving the base link of the robot. Should be 1 for new development, and 0 when running old test cases.
`panelSection *truss` - the truss
`ffController *ffc` - the controller used during simulation
`relativePath *p` - the relativePath used for all trajectories. The path's waypoints are set by the trussPath, and by several methods described below
`triple trussInitialX` - the initial position of the panelSection's origin.
`quaternion trussInitialQ` - the initial orientation of the panelSection.
`endPointRec *fixedPt, *movingPt` - the end effectors that are currently fixed and moving, respectively.
`endPointRec, *baseEndPt, *toolEndPt` - the end effectors on the base and distal links of the robot, respectively.

New methods

`void updatePhase(int newPhase)` - sets `phase`, `fixedPt`, and `movingPt` based on `newPhase`.
`int walkAlongTruss(trussPath *tp, int doInitialMove)` - simulates the robot walking along the truss, using the supplied `trussPath` to generate gait trajectories. If `doInitialMove` is 1, then the robot will first grasp the truss with both grippers, with a separation of 0.6m between grippers. Otherwise, the robot will immediately begin walking from its current pose. Currently, `walkAlongTruss` is always called with `doInitialMove` set to 1.
`int followPath(relativePath *p, int useDynamics = 1, int useMetrics = 1)` -
`void initializeCfgAndTrussPoses(void)` - Sets the initial pose of the truss and of the configuration so that the gripper indicated by `firstGraspPoint` is grasping the truss rod 0.3m from its beginning.
`void initializePath(relativePath *p, int newPhase)` -
`int initializeTrussPath(relativePath *p,trussPath *tp, int newPhase)` - computes a trajectory for the next gait cycle and assigns the trajectory to the appropriate end effector.
`int initializeTransitionPath(relativePath *p, int newPhase, int type, int side, double idx, double l, double twist, double roll)` - computes a trajectory for the gripper indicated by `newPhase` to move to the position given by (`l, twist, roll`) on the truss rod indicated by {`type, side, idx`}. See `panelSection` for interpretation of the parameters.
`int initializeRotationPath(relativePath *p)` - computes a trajectory causing the robot to rotate 180 degrees about the current truss segment and assigns the trajectory to the appropriate end effector.

`int initializeInspectionPath(relativePath *p)` - computes a trajectory for the inspection phase of the task and assigns it to the appropriate end effector.

`void convertToRootCoords(triple &p, quaternion &q)` - converts `p` and `q` from a via point for the base link to a via point for the distal link. This allows trajectories to always be computed in world coordinates, and then transformed appropriately so that the base link can move along the trajectory.

p-file variables

Note that although the `walkerEvaluator` is derived from the `pathEvaluator`, originPos and originOrientation are the only p-file variables from pathEvaluator used by walkerEvaluator. While values must still be supplied for the other required `pathEvaluator` p-file variables, the values are not used.

Variable	Description	Default value
double walkingWeight	See "Data members" above.	
double transitionWeight	See "Data members" above.	
double approachDist	See "Data members" above.	
double transitionDist	See "Data members" above.	
double inspectionStandoff	See "Data members" above.	
double stride	See "Data members" above.	
double viaTol	See "Data members" above.	
double angleTol	See "Data members" above.	
double tol	See "Data members" above.	
double vTol	See "Data members" above.	
double hOffset	See "Data members" above.	
double vOffset	See "Data members" above.	
double approachAngle	See "Data members" above.	
double pathTimeout	See "Data members" above.	
int firstGraspPoint	See "Data members" above. Must be 0 or 1.	
int computeLinkPath	See "Data members" above.	0
int stopAfterWalking	See "Data members" above.	0

Task parameters

Variable	Description
double approachDist	See "Data members" above.
double transitionDist	See "Data members" above.
double inspectionStandoff	See "Data members" above.
double stride	See "Data members" above.
double viaTol	See "Data members" above.
double angleTol	See "Data members" above.
double tol	See "Data members" above.
double vTol	See "Data members" above.
double hOffset	See "Data members" above.
double vOffset	See "Data members" above.
double approachAngle	See "Data members" above.
int firstGraspPoint	See "Data members" above. Only the least significant bit of the parameter's ival is used.
int computeLinkPath	See "Data members" above. Only the least significant bit of the parameter's ival is used.

Source file

```
apl/walker/walkerEval.*
```

Dynamic Libraries

```
libsynPMWalker.so
libsynEvalWalker.so
libsynEvalWalkerD.so
```

See also

```
evaluator
pathEvaluator
evComponent
panelSection
```

ffEvaluator

Derivation

```
ffEvaluator : pathEvaluator : evaluator : synObject
```

Data members

New methods

p-file variables

Task parameters

Source file

```
apl/ff/ffEvaluator.*
```

See also

```
evaluator
pathEvaluator
evComponent
```

evComponent

The evComponent class is a base class for modular simulation objects and capabilities such as trajectories, controllers, payloads, and collision detection algorithms. The evComponents' modular nature and standardized interface allows simulation capabilities to be specified at runtime through the use of setup files, rather than requiring a simulation to be hard-coded.

The evaluator class provides bookkeeping and high-level simulation control, but relies entirely on evComponents for detailed simulation algorithms. The evaluator calls each evComponent's methods in a standard order, allowing subclasses of evComponent to perform appropriate initialization and interface with the evaluator and with other evComponents in a regular manner. The evComponent methods are called as follows:

int readParams(paramParser *parser) - called immediately after creating the evComponent

int evInit(evaluator *ev) - called after all evComponents have been created; evComponents can use ev->getEvComponent...() to find other evComponents for interfacing.

The following methods are called each time a configuration is evaluated:

`int init(configuration *cfg)` - before simulation

`int setVariables(ptrList *taskParamRecs)` - after init, before simulation

`int cleanup()` - after simulation, before the configuration is deleted.

Subclasses of `evComponent` will usually override these virtual functions and others listed under "New methods" below. The argument to `setVariables`() is a ptrList containing pointers to `taskParamRecords`. The members of the `taskParamRecord` that are useful to the `evComponent` are:

`const char *varName` - the name of the variable represented by the task parameter

`param *p` - the value of the variable. `evComponents` can use either the integer value (`p->ival`) or real value (`p->val`).

Finally, the `evComponent` defines another virtual function, `update`(). This method is called at each simulation time step, allowing the `evComponent` to continuously update state variables or perform other simulation needs.

Derivation

```
evComponent : synObject
```

Data members

`evaluator *ev` - the `evaluator`, passed as an argument to `evInit`()
`configuration *cfg` - the configuration currently being simulated
`const char *label` - a symbolic label (possibly NULL), normally read from the p-file
`int active` - indicates whether the `evComponent` is currently being used. The `update`() method of the `evComponent` will not be called if `active` is 0.

New methods

`virtual int readParams(paramParser *parser)` - reads parameters that are independent of each configuration. In derived classes, this method should call the base class's `readParams()` method, and should return 0 if the base class's method fails. Returns 1 on success, 0 on failure.
`virtual int setVariables(const ptrList *taskParamRecs)` - reads value for variable indicated by `varName` from parameter `p`. In derived classes, this method should call the base class's `setVariables()` method, and should return 0 if the base class's method fails. Returns 1 on success, 0 on failure.
`virtual int evInit(evaluator *Ev)` - Performs initialization dependent on `ev`. This method should call base class's `evInit`() function before performing class-specific initialization, and should return 0 if the base class's method fails. This function is called once after all `evComponents` have been created, and allows `evComponents` to interface to each other (since `ev` contains a list of `evComponents`).
`virtual int init(configuration *Cfg)` - Performs initialization dependent on `cfg`. This function is called each time a new configuration is evaluated. In derived classes, this should call base class's `init`() function *before* performing class-specific initialization, and should return 0 if the base class's method fails. Returns 1 on success, 0 on failure.
`virtual int cleanup(void)` - Performs configuration-dependent cleanup, such as freeing memory allocated by `init`() or `update`(). This method should call the base class's `cleanup`() method *after* performing class-specific cleanup, and should do nothing if cfg is NULL. Do not call this from a derived class's destructor; it will be called before the destructor, after each configuration has been evaluated. This method should return 0 if the base class's method fails. Returns 1 on success, 0 on failure.
`virtual int update(void)` - This function is called each time step of simulation to allow the `evComponent` to update state-dependent variables.

p-file variables

Variable	Description	Default value
int enabled	Determines whether or not the evComponent is used. Note that this is actually read by the evaluatorbefore creating the evComponent.	1

Variable	Description	Default value
int verboseLevel	Sets the level of text output.	1 (SO_WARNINGS)
char label[]	A symbolic label for the evComponent, allowing it to be found by other evComponents.	

Source file

`db/evComponent.h`

See also

`evaluator`

collisionDetector

The collisionDetector is an evComponent that uses the RAPID collision detection library to detect robot self-collisions, and collision between the robot or its payload with external obstacles. The collisionDetector can also compute the points of intersection between polyhedra when being used for simulation of contact. Some modules define low-detail polyhedra for their links; the collision detector will use these low-detail models if they have been defined. Several behaviors for detecting self-collisions are available, based on the value of checkSelfCollisions:

CD_ALL (0)- all intersections between different links of the robot should be reported

CD_IGNORE_JOINTS (1)- ignore intersections between links that are directly connected by a joint. This is useful when using joint modules that do not have a small gap between links.

CD_NONE (2) - no self-collisions will be reported.

Additionally, modules may override the virtual function `addAdditionalIgnoredCDPairs()` to indicate that specific links should be allowed to intersect. An example of the use of this method is the `prismaticTube` module, which uses cylinders rather than hollow tubes for its low-detail representation to reduce polygon count. The `prismaticTube` allows these cylinders to intersect each other so that they do not generate collisions. To allow the links containting parts p1 and p2 to intersect, a module's `addAdditionalIgnoredCDPairs()` method should contain the following statements:

```
p1->l->cdb->addIgnoredObject(p2->l->cdb);
p2->l->cdb->addIgnoredObject(p1->l->cdb);
```

Derivation

```
collisionDetector : evComponent : synObject
```

Data members

`static int currentTriId` - a counter used to generate unique IDs for RAPID triangles.

`int numCollisions` - the number of intersecting bodies last found by `checkCollisions()`.

`int checkSelfCollisions` - determines how detection of self-collisions is performed (see above).

`int checkJointLimits` - if 1, then a collision will be generated any time a joint moves past its limit. This is useful when dynamic simulation is being used, but without unilateral constraints to enforce joint limits.

`int createITris` - a value of 1 indicates that triangles should be allocated for purposes of polyhedra intersection calculation. This variable must be 1 if computeIntersections() will be called.
`int addCfgLinks` - 1 indicates that a configuration's links should be checked for collisions. When 0, it is assumed that some other evComponent or the evaluator will specifically indicate which links should be checked via the `addLinkModel()` method.
`ptrList *cfgObjects` - a list of `cdObjects` that represent parts of the configuration or attached payloads
`ptrList *obstacles` - `cdObjects` that represent external obstacles
`ptrList *objects` - the union of `cfgObjects` and `obstacles`
`ptrList *pairs` - a list of `cdPairs`, each of which contains a pair of `cdObjects` that should be checked for collisions.

New methods

Checking collisions and intersections

`int checkCollisions(void)` - computes and returns the number of objects that are intersecting. Also stores this number in numCollisions.
`ptrList *computeIntersections(void)` - returns a `ptrList` of `cdObjectIntersections`, one for each pair of objects that are intersecting. Each `cdObjectIntersection` contains a list of intersection points between two triangles, one from each object. The caller of this method should call `collisionDetector::freeIntersectionList`, with the returned list as the argument, when done with the contents of the list.

Adding and removing objects

`int addLinkModel(link *l, void *userData = NULL)` - adds a new `cdBody` for the specified link, and sets the `cdBody's userData` field to the supplied value.
`int createLinkObstaclePairs(void)` - when using `addLinkModel`, the caller should call this function after adding all links.
`void addObject(cdObject *o)` - adds the supplied object to the list of objects and creates new cdPairs if necessary.
`void addObjects(ptrList *p)` - same as `addObject`, but for a list of objects
`int removeObject(cdObject *o)` - removes the cdObject and any pairs that may reference it.
`void addObstacle(cdPolyhedronObstacle *o)` - adds an obstacle.

Convenience functions

`cdObject *getObject(int i)` - returns the i^{th} `cdObject`.
`cdPair *getPair(int i)` - returns the i^{th} `cdPair`.
`void freeIntersectionList(ptrList *ilist)` - deletes the list returned by computeIntersections, and its contents.

p-file variables

Variable	Description	Default value
int checkSelfCollisions	See ‘Data members’ above	-
int readObstacles	See ‘Data members’ above	-
int checkJointLimits	See ‘Data members’ above	-
int createITris	See ‘Data members’ above	0
int addCfgLinks	See ‘Data members’ above	1
char obstacleFilename[]	If readObstacles is 1, this variable is read to determine the name of the file containing obstacle definitions.	-

Source file

```
collision/collision.*
```

See also

```
evaluator
evComponent
```

genericPath

The genericPath is an abstract base class for representing trajectories for robot end effectors. Tool forces and torques to be applied along the path may be specified, and a payload to be moved along the path may also be specified.

Derivation

```
genericPath : evComponent : synObject
```

Data members

Variables controlling path behavior

`int pathNum` - provides an ordering for paths; for example, the `pathEvaluator` and kin assign the path to the path group indicated by this number.

`int endPtNum` - the index of the end effector for which the path is intended

`int usePayload` - 1 if the path has an associated payload, 0 otherwise.

`payload *pl` - a pointer to the payload used for the path.

`int useAppliedForce` - 1 if `toolForce` and `toolTorque` should be exerted by the end effector during the path, 0 otherwise.

`triple toolForce` - the force (in world coordinates) to exert along the path.

`triple toolTorque` - the torque (moment, really, in world coordinates) to exert along the path.

`endPointRec *endPt` - a pointer to the end effector following the path. This gets set by `configuration::setToolPath`.

`const char *payloadLabel` - when `usePayload` is 1, this label specifies which payload from the evaluator's p-file is used.

`int alignPayload` - indicates whether or not the payload's connector should be aligned to the end effector's TCP before attaching the payload to the end effector. A value of 1 indicates that the payload's pose should be altered to bring it into alignment, while 0 indicates that the payload should be rigidly attached to the end effector from its current pose.

`int payloadConnectorID` - the connector ID for the connector on the payload that should be aligned to the end effector's TCP if `alignPayload` is 1.

`triple lastVel` - the linear velocity (in world coordinates) of the TCP at the previous time step. This is normally set by `sriController` or a derived class.

`triple lastOmega` - the angular velocity (in world coordinates) of the TCP at the previous time step. This is normally set by `sriController` or a derived class.

New methods

`virtual void getStartPoint(triple &t, quaternion &q)` - (pure virtual) Returns the position and orientation of the first point on the path

`virtual vector computeVel(double dt, quaternion *outputFrame = NULL)` - (pure virtual) Returns a 6-vector containing the desired linear and angular velocity of the end effector. `dt` is the expected size of the time step that will be taken using the returned velocity command, and is used for computing the acceleration. If `outputFrame` is non-null, then the velocities are transformed by the supplied quaternion.

`virtual void reset(void)` - resets all state variables to their initial values.

`virtual int atStart(void)` - returns 1 if the end effector has reached the starting point of the trajectory.

`virtual int done(void)` - returns 1 if the end effector has reached the end of the path

`virtual double completion(void)` - returns a number between 0 and 1 indicating the portion of the path that has been completed.

`virtual triple computeCrossTrackError(triple &pos)` - (pure virtual) Computes the cross-track error (distance to the closest point on the trajectory, in world coordinates) of `pos`.

`virtual triple computeOrientationError(quaternion &q)` - (pure virtual) - Returns the rotational error (in axis*angle format) between the supplied orientation and the current goal or location on the path.

p-file variables

Variable	Description	Default value
int endPtNum	See “Data Members” above.	
int pathNum	See “Data Members” above. This can be set to 0 if the `evaluator` does not use it.	
int usePayload	See “Data Members” above. Note that if a value of 0 is specified in the p-file, the `evaluator` or an `evComponent` can still set `usePayload` to 1 later if the value of `pl` is also set.	0
int useAppliedForce	See “Data Members” above.	0
int payloadConnectorID	See “Data Members” above.	
int alignPayload	See “Data Members” above.	
char payloadLabel[]	See “Data Members” above.	
double toolForce{X,Y,Z}	See “Data Members” above. The `evaluator` or an `evComponent` can set this later.	0
double toolTorque{X,Y,Z}	See “Data Members” above. The `evaluator` or an `evComponent` can set this later.	0

Source file

```
ctrl/path.*
```

See also

```
evaluator
evComponent
```

path

The path class represents trajectories for a robot's Tool Control Point (TCP) as a series of via points (positions and orientations) in world coordinates. Via points may either be read from a file, or set by an evaluator or evComponent. Parameters for the velocity profile to use when moving between points may be specified through a p-file or through task parameters, and the TCP can either stop at or continue through each via point. The user can also specify the position and velocity tolerances used to determine when the TCP is considered to have reached a via point.

A path is usually used in conjunction with the sriController or subclasses thereof, and interfaces with the controller primarily through the computeVel method. computeVel considers the position, orientation, and linear and angular velocity of the TCP and computes a desired velocity for the TCP that moves it directly towards the next via point on the path. The TCP's acceleration and deceleration are bounded by maxAcc and maxOmegaDot, and the computed velocity command is bounded by maxVel and maxOmega. PID control is used to compute velocity commands in the vicinity of via points, with the PID gains computed from the velocity and acceleration parameters. The ability of the TCP to stop at a via point (i.e. the convergence) is heavily dependent on the velocity and acceleration parameters, the position and velocity tolerances for the via point, and the simulation stepsize used. Including the velocity and accleration parameters as task parameters is highly recommended.

The path class defines several virtual functions that subclasses can use to provide new behavior. getPoint() returns the ith via point, allowing a subclass to transform or otherwise manipulate the via points stored in waypt[] and orientation[] before they are used. modifyEndpointPoseAndPath() also allows subclasses to modify the TCP and via point locations used by computeVel(), and modifyVelocityCommand() can be used by subclasses to modify the velocity command computed by computeVel before it is returned to the caller.

Derivation

```
path : genericPath : evComponent : synObject
```

Data members

Waypoint information

`int np` - the number of via points in the path

`triple *waypt` - position for each via point.

`quaternion *orientation` - orientation for each via point

`int *stopAtPoint` - an array of `np` ints indicating whether the end effector should stop at each via point.

`int useSameTol` - a value of 1 implies that the same tolerances should be used for all via points; 0 implies that a tolerance will be supplied for each via poitn.

double *tol - distance from via point at which the end effector is considered to have reached the via point. If `useSameTol` is 1, then tol will point to a double; if useSameTol is 0, then tol is an array of `np` doubles.

`double *vTol` - when stopping at a via point, the end effector's velocity must be less than vTol. Either an array or a single value, depending on useSameTol (see tol above).

`double *angleTol` - angle from via point orientation at which the end effector is considered to have reached the via point. Either an array or a single value, depending on `useSameTol` (see `tol` above).

Velocity and acceleration variables.

`double vel` - Maximum linear velocity

`double omega` - Maximum angular velocity

`double maxAcc` - Maximum linear acceleration

`double maxOmegaDot` - Maximum angular acceleration

`double KImultiplier` - When greater than zero, this number muliplies `kp` (below) to determine `ki`.

Control gains. These are normally computed from vel, omega, macAcc, and maxOmegaDot.

`double kv, kp, ki` - PID gains for position

`double qkv, qkp` - PD gains for orientation

`double pdThresh` - distance from via point at which to start PID control.

`double qpdThresh` - angle from via point orientation at which to start PID control

State variables.

`int currentPt` - the index of the most recently reached via point

`int targetPt` - the index of the point currently being moved towards

`double targetDist` - the distance to the target point

`triple ep` - accumulated position error vector, used with `ki`

File formats

When the p-file variable `readPath` is set to 1, the path object will read via point information from the file indicated by `pathFilename`. The format of the path file depends on the value of useSameTol, as specified in the p-file: if useSameTol is 0, then each waypoint must include data for the tolerances to be used for each point; otherwise, the tolerances specified in the p-file are used. In either case, the path file is a text file containing a set of numbers for each via point:

- an integer for `stopAtPoint` for the via point
- three floating-point numbers for the location of the via point
- four floating-point numbers for the quaternion representing the TCP orientation at the via point
- and, when useSameTol is 0, three floating point numbers for `tol`,

angleTol, and vTol for the via point.

Thus, the path file for a path with two points and useSameTol equal to 1 might look like this:

```
1
0 -0.5 1.5
1 0 0 0

1 0.5 1.5
1 0 0 0
```

If useSameTol were 0, then each point would be followed by an additional three numbers for the tolerances.

New methods

`virtual int getTargetPoint(triple &pos, quaternion &q, triple &targetPos, quaternion &targetQ)` - returns the index of the target point given the current TCP position pos and orientation q, and returns the position and orientation of the target point in targetPos and targetQ.

`virtual void getPoint(int i, triple &pos, quaternion &q)` Returns the position and orientation of the ith via point in pos and q, respectively. Subclasses may override this function, which is used by getTargetPoint.

`virtual void modifyEndpointPoseAndPath(triple &pos, quaternion &q, triple &tpos, quaternion &tq)` This virtual function does nothing by default, but is provided so that subclasses can modify the current TCP coordinates before they are used by computeVel. For example, the relativePath overrides this method and uses it to convert the TCP to a different coordinate system.

`virtual void modifyVelocityCommand(triple &v, triple &omega)` - This method does nothing by default, but is provided so that derived classes can modify the command generated by computeVel. The relativePath overrides this method and uses it to compute s world-space velocity command.

`virtual void setNumPoints(int n, int useSameTol)` - resizes the arrays for via point information to the desired size.

p-file variables

Variable	Description	Default value
int np	See “Data members” above.	
double vel	See “Data members” above.	
double omega	See “Data members” above.	
double maxOmegaDot	See “Data members” above.	
double maxAcc	See “Data members” above.	
double KImultiplier	See “Data members” above.	-1
int useSameTol	See “Data members” above.	1
double tol	See “Data members” above. If useSameTol is 1, this value must be supplied.	
double vTol	See “Data members” above. If useSameTol is 1, this value must be supplied.	
double angleTol	See “Data members” above. If useSameTol is 1, this value must be supplied.	
int readPath	Indicates whether the via points should be read from a file.	1
char pathFilename[]	If readPath is 1, this variable should be set to the name of the file containing the via points	

Task parameters

Variable	Description
double vel	See “Data members” above
double omega	See “Data members” above
double maxOmegaDot	See “Data members” above
double maxAcc	See “Data members” above

Source file

```
ctrl/path.*
```

See also

```
evaluator
evComponent
```

```
genericPath
payload
pathEvaluator
```

relativePath

The relativePath is similar to the path class from which it is derived, with the main difference being that the via points are defined relative to a link's coordinate system rather than world coordinates. The link is normally either the robot's base link or a payload, though any link may be used. The relativePath achieves this by overriding the path methods getPoint, modifyEndpointPoseAndPath and modifyVelocityCommand, as well as the evComponent method update. The former three convert between world and link coordinates.

Derivation

```
relativePath : path : genericPath : evComponent : synObject
```

Data members

link *l - the link whose coordinate system the via points are defined in
payload *relPayload - if non-NULL, the payload containing l
const char *relativePayloadLabel - the label of the payload (if any) which the path is relative to
triple btoolForce, btoolTorque - the applied force and torque relative to l. These are copied from toolForce and toolTorque during evInit, and can be set manually. update() converts these to world coordinates and stores them in toolForce and toolTorque.
int computeLinkPath - if 1, the velocity command should be modified so that l's center of mass follows a straight line in world coordinates.

p-file variables

Variable	Description	Default value
relativePayloadLabel	If the path is relative to a payload and automatic initialization is desired, this variable should be set to the label of the payload. If this variable is not defined, it is assumed that the evaluator or an evComponent will set l.	NULL
computeLinkPath	See "Data members" above	0

Source file

```
ctrl/path.*
```

See also

```
evaluator
```

```
evComponent
path
```

payload

The payload class represents rigid-body payloads composed of multiple polyhedra. Each polyhedron can have a separate density and color (for display), and the payload can have multiple coordinate frames representing grasp points for a tool's TCP.

Derivation

```
payload : evComponent : synObject
```

Data members

`link *l` - the link representing the payload's geometry
`cdBody *b` - a `cdBody` for `l`, if collsion detection is being used
`collisionDetector *cd` - the collision detector used by the evaluator, if any. Set automatically by `evInit()`.
`ptrList *connectors` - list of connectors
`triple xrel` - position relative to TCP; computed by `computeToolCoords()`
`quaternion qrel` - orientation relative to TCP; computed by `computeToolCoords()`
`triple originPos` - initial position of `l`'s origin
`quaternion initialQ` - initial orientation of `l`

New methods

Initialization

`virtual int readFromFile(void)` - returns 1 if `initFromFile()` should be called. Derived classes can override this if they do not need to read geometry from a file.

`int initFromFile(const char *filename, const float *color)` Reads payload geometry from the specified file. If `perPolyColor` is 0, color can be used to supply an array of floats representing a color in RGB format (with each component between 0 and 1).

Aligning to TCP

`void computeToolCoords(triple &xtool, quaternion &qtool)` - Given the payload's current pose and tool pose (`xtool`, `qtool`), compute the pose (`xrel`, `qrel`) relative to the tool

`void updatePose(triple &xtool, quaternion &qtool)` - updates l's pose given the new TCP location (`xtool, qtool`). Also compute's `l->I` from `l->Ibody`.

`int alignConnectorToPose(triple &xtool, quaternion &qtool, int cnId)` - sets `l`'s pose to align the connector given by `cnId` with (`xtool`, `qtool`)

Convenience functions

`int numConnectors(void)` - returns the number of connectors>n; }
`connector *getConnector(int n)` - returns the specified connector
`connector *getConnectorById(int id)` - returns the connector with specified ID

File format

The payload file format specifies one or more convex polyhedra, each with a separate material density. The payload file also describes any coordinate frames attached to the payload, for use by controllers. The format is:

```
<number of coordinate frames>
<coordinate frame 1: 16 floating point numbers specifying a
     homogeneous coordinate system>
<other coordinate frames, in same format as the first>

<number of polyhedra>

for each polyhedra:
  <density> in kg/m^3
  <r g b> (0->1 for each component;
           omit if perPolyColors = 0)
  <number of points>

  <list of points: x, y, and z for each point>

  <number of faces>

  for each face:
    <number of vertices>
     <list of point indices from the list of points (indices
      start at 0)>
```

p-file variables

Variable	Description	Default value
double originPos{X,Y,Z}	See “Data members” above	0
double initialQ{R,I,J,K}	See “Data members” above. Automatically normalized.	0
int perPolyColors	If 1, indicates that a color is stored in the payload file for each polyhedron	0

Variable	Description	Default value
float color{R,G,B}	If perPolyColors is 0, these variables supply the color for the payload.	{1, 1, 0} (yellow)

Task parameters

Variable	Description
double origin{X,Y,Z}	See `originPos` in “Data members” above.
double initial{R,I,J,K}	See `initialQ` in “Data members” above. Automatically normalized.

Source file

```
cfg/payload.*
```

See also

```
evaluator
evComponent
path
```

panelSection

Derivation

```
payload : evComponent : synObject
```

Data members

New methods

p-file variables

Variable	Description	Default value

Task parameters

Variable	Description	Default value

Source file

```
cfg/payload.*
```

See also

```
evaluator
evComponent
path
```

DEsolver

Derivation

```
DEsolver : evComponent : synObject
```

Data members

New methods

p-file variables

Variable	Description	Default value

Task parameters

Variable	Description	Default value

Source file

```
ctrl/solver.*
```

See also

```
evaluator
evComponent
```

rungeKutta4

Derivation

```
rungeKutta4 : DEsolver : evComponent : synObject
```

Data members

New methods

p-file variables

Variable	Description	Default value

Task parameters

Variable	Description	Default value

Source file

```
ctrl/solver.*
```

See also

```
evaluator
evComponent
DEsolver
```

controller

Derivation

```
controller : evComponent : synObject
```

Description

Data members

New methods

p-file variables

Variable	Description	Default value

Task parameters

Variable	Description	Default value

Source file

```
ctrl/controller.*
```

See also

```
evaluator
evComponent
DEsolver
```

sriController

The `sriController` uses the Singularity Robust Inverse (SRI) of a robot's Jacobian to cause the robot's end effectors to follow Cartesian-space (rather than joint space) trajectories. The `sriController` is usually used in conjunction with the path and relativePath classes, which represent Cartesian trajectories. See Section 4.3 in *Darwin2K: An Evolutionary Approach to Automated Design for Robotics* for more information about how the Singularity Robust Inverse is used. The `sriController` will generate joint velocity commands for any degree-of-freedom that affects the motion of an end effector that currently has a path assigned to it.

When a robot has redundant degrees of freedom, the sriController can use them to attempt to avoid joint limits. It does this by projecting the vector returned by `configuration::computeLimitGradient()` onto the nullspace of the Jacobian. The projected vector will cause the robot's joints to move away from their limits, but will not affect the instantaneous velocity if the end effector(s). The variable `doGradientDescent` indicates whether this should be done; when equal to 1, any time the gradient vector is longer than `ignoreLimitThresh`, the gradient will be projected onto the nullspace, multiplied by `gradientStepSize`, and added to the joint velocity command computed from the SRI. These parameters can affect the ability of the controller to precisely stop at trajectory via points, so it is recommended that they be included as task parameters when performing synthesis or optimization.

Derivation

```
sriController : controller : evComponent : synObject
```

Data members

Note: only those data members that must be set by users of the class are noted here, as there are many internal variables.

`double minRate` - theshhold above which generalized velocity vector (concatenation of linear and angular velocity of all endpoints) is considered non-zero; used for determining whether the robot has become "stuck"

sriControllerMode mode - determines how the robot's DOFs are used:

- `sriController::FIND_BASE_POSE` - only base degrees of freedom are used
- `sriController::FIND_MECH_POSE` - all degrees of freedom are used, and if the robot has a non-holonomic base, the base is allowed to move freely
- `sriController::NORMAL` - the robot's DOFs operate normally.

SRI parameters

`double mu` - ratio of current to previous manipulability that triggers use of SRI rather than pseudo-inverse

`double lambda0` - multiplier for the identity matrix

Joint limit avoidance and redundancy optimization

`int doGradientDescent` - 1 if the redundant DOFs should be used to avoid joint limits, 0 otherwise.

`double ignoreLimitThresh` - gradients with norms less than this will be ignored.

`double gradientStepSize` - multiplier for the gradient

New methods

`void setMode(sriControllerMode m)` - sets the current mode of the controller.

p-file variables

Variable	Description	Default value
double mu	See “Data members” above	0.1
double lambda0	See “Data members” above	0.0003
double minrate	See “Data members” above	0.0001
double ignoreLimitThresh	See “Data members” above	0.1
double gradientStepSize	See “Data members” above	0.001
int doGradientDescent	See “Data members” above	1

Task parameters

Variable	Description
double mu	See “Data members” above
double lambda0	See “Data members” above
double ignoreLimitThresh	See “Data members” above
double gradientStepSize	See “Data members” above
int doGradientDescent	See “Data members” above. Only the least-significant bit is used.

Source file

`ctrl/sriController.*`

See also

```
evaluator
evComponent
controller
DEsolver
```

ffController

The `ffController` is similar to the `sriController`, but uses the robot's dynamic model to account for the reaction forces applied to free-flying bases by a robot's manipulator(s). The programming and file interfaces for the `ffController` are identical to those of the `sriController`. See Section 4.4.4 in *Darwin2K: An Evolutionary Approach to Automated Design for Robotics* for a derivation of the `ffController's` behavior.

Derivation

```
ffController : sriController : controller : evComponent
: synObject
```

Source file

```
ctrl/ffController.*
```

See also

```
evaluator
evComponent
DEsolver
sriController
```

pidController

Derivation

```
pidController : controller : evComponent : synObject
```

Data members

New methods

p-file variables

Variable	Description	Default value

Task parameters

Variable	Description	Default value

Source file

```
ctrl/pidController.*
```

See also

```
evaluator
evComponent
DEsolver
controller
```

roverController

Derivation

```
roverController : controller : evComponent : synObject
```

Description

Data members

New methods

p-file variables

Variable	Description	Default value

Task parameters

Variable	Description	Default value

Source file

```
ctrl/roverController.*
```

See also

```
evaluator
evComponent
DEsolver
controller
```

Glossary

actuator saturation - the ratio of applied torque (or force) to an actuator's maximum available torque (or force). An actuator saturation greater than one indicates that the controller commanded a torque beyond the actuator's capability.

class - in object-oriented programming, a class is a description of a data type and a set of related functions.

component context - a set of component lists, with one list for each of a module's component-selection parameters. Each component list contains one or more components of similar type, e.g. motors or actuators.

configuration - the general form of the robot, including kinematics and other geometry at the bare minimum and usually including descriptions of inertial properties, actuator and material selection, and structural geometry.

configuration graph - see Parameterized Module Configuration Graph (PMCG)

configuration optimization - the process of improving the performance of a robot configuration (or small number of configurations) through parametric variation

configuration synthesis - the process of generating a high-level description of a robot and improving its performance through parametric and topological variation.

`const` flag - a flag associated with parameters and attachments in the Parameterized Module Configuration Graph that can be set by the designer to indicate that the parameter or attachment should not be changed by the synthesizer.

degree of freedom (DOF) - an independent variable describing part of the state of a robot. The state of an *n*-DOF robot can be uniquely and completely described by *n* variables.

Denavit-Hartenburg (D-H) parameters - a commonly-used set of parameters describing the kinematics of a robot, in which each pair of successive joints is characterized by a distance between joint axes *a*, a twist between joint axes α, an offset *d*, and a joint angle θ. See e.g. [Denavit55] for details.

directed acyclic graph (DAG) - a graph in which each edge has a direction and in which no path along the edges passes through a node more than once.

dynamic simulation - computing the motion of a mechanical system (e.g. a robot) based on the forces and torques applied to the system. (Compare to kinematic simulation.)

elitism - in an evolutionary algorithm, elitism refers to explicitly preserving or reproducing a subset of the population that are considered to be 'best'.

elite set - in an evolutionary algorithm, the subset of the population that is considered to

be 'best'. In a single-objective EA, the elite set might contain the *n*-best solutions; in a multi-objective EA, it might contain the Pareto-optimal set. In Darwin2K, it is defined to be the feasibly-optimal set.

evolutionary algorithm (EA) - an optimization algorithm (often probabilistic in nature) based on biological theories of evolution. Typical features include the use of a population of solutions, selection of solutions based on fitness (analogous to "survival of the fittest"), and creating new solutions by combining or modifying existing solutions.

evolutionary synthesis engine (ESE) - in Darwin2K, the program that maintains a population of configurations, creates new configurations, and sends them to one or more evaluation processes which measure their performance.

feasibly-optimal set - in Darwin2K, the subset of the population that is considered 'best'. If any configurations are feasible, then the feasibly-optimal set is the Pareto-optimal set taken over all feasible configurations in the population. If no configurations are feasible, then the feasibly-optimal set is the Pareto-optimal set taken over the entire population.

fitness - a figure of merit reflecting the performance of a solution in an evolutionary algorithm

fitness-proportionate selection - in an evolutionary algorithm, a method of selecting solutions for reproduction in which a solution is selected with probability directly proportional to its fitness

generational genetic algorithm - a genetic algorithm that creates an entire new population of solutions at once and then evaluates them all. This is the standard method for creating new solutions in a GA.

genetic algorithm (GA) - an evolutionary algorithm that represents solutions as a string of symbols (usually a fixed-length string of bits), uses crossover, duplication, and mutation to create new solutions, and selects solutions for reproduction on the basis of their performance.

genetic programming (GP) - an evolutionary algorithm that represents solutions as programs. The fitness of a program is determined by executing it, rather than by measuring some property of the program.

globally optimal planner - a planner that is guaranteed to find the globally-optimal motion with respect to a cost metric (usually either time or distance) down to the level of discretization used.

graph - a data structure or mathematical object consisting of nodes connected by edges. Each edge has two endpoints and may have a direction associated with it.

kernel - an initial configuration specified by the designer from which all other configurations are generated through the application of genetic operators.

kinematic simulation - computing the motion of a mechanical system (e.g. a robot) by considering only position variables and their derivatives, rather than forces and torques.

kinematics - the study of a mechanism's motions without regard to forces, torques, or inertia.

locally optimal planner - a planner that is not guaranteed to find the optimal motion. Typically much less computationally expensive than globally-optimal planners.

member - in OOP, part of the description of a class. A data member describes a property of objects that belong to the class, while a member function (or method) describes the behavior of objects belonging to the class.

method - see member

module - a self-contained software object representing part of a robot. In Darwin2K, a module contains data describing the its properties, and functions describing its behavior.

object - in Object-Oriented Programming, a self-contained set of properties and associated procedures.

parameter - in Darwin2K, a value that may be varied by the synthesis algorithm. Parameters are described by several features: the minimum and maximum values of the parameter, the resolution, an integer value, a real value, and a `const`-flag. Modules can have parameters, and configurations can have parameters associated with them that describe aspects of the task.

parameterized module - an object representing part of a robot, including both data and function members. Parameterized modules may have zero or more parameters describing arbitrary properties, and can specify connectors that indicate how the module can be connected to other modules.

parameterized module configuration graph (PMCG) - the representation for robot configurations used in Darwin2K. The PMCG consists of a list of modules and connections between them and allows both parametric and topological variation of robot properties.

Pareto-optimal set - given a set of multi-dimensional measurements (e.g. robot configurations with performance measurements), the Pareto-optimal set is those measurements that are better than or equivalent to every other measurement along at least one dimension. Also called the non-inferior set, as every member of the set is not inferior (i.e. worse than another element in all dimensions) to any other element of the set.

simple genetic algorithm (SGA) - the basic genetic algorithm, using a single population and representing solutions as fixed-length bit strings.

steady-state genetic algorithm (SSGA) - a genetic algorithm that continuously adds solutions to, and removes solutions from, a population. Contrast to a generational genetic algorithm.

task parameter - a variable or property that does not belong to a configuration but which can be optimized by the synthesizer. Examples include via point location and controller gains.

tool control point (TCP) - a position and orientation defined relative to an end effector (tool) that is used to specify the motion of the effector. For example, the TCP for a gripper might be the point midway between the gripper's fingers.

Index