Beast II SDK: version 0.2

Remco R. Bouckaert
remco@cs.{auckland|waikato}.ac.nz
Department of Computer Science
University of Auckland & University of Waikato

August 2, 2011

Abstract

This is a short description of the Beast II Software Development Kit, which includes a Java library for building Markov Chain Monte Carlo (MCMC) applications using the Metropolis Hastings method and a library for Bayesian analysis of evolutionary problems.

In particular, there is support for efficient updating of models, GUIs for building models and support for documentation. Newly written Plugins will be directly available in the GUIs, online help and HTML documentation.

1 Introduction

Beast II is written in Java, open source and licensed under the Lesser GNU Public License. The Beast II SDK can be downloaded from `http://code.google.com/p/beast2/downloads/list` and the source is available from `http://code.google.com/p/beast2/source/checkout`.
Beast II typically runs as a standalone application, started from the command line with `java -jar beast.jar` which starts `beast.app.BeastMCMC`. An XML file should be specified as command line argument. XML files are used to store models and data in a single place. See Section 4 for details.
To use the SDK, you write Java classes that derive from the `Plugin` class, or derive from any of the more specialized classes that derive from `Plugin`. By default, the classes are expected to reside in a jar file in the `beastlib` directory from where BeastMCMC (or any of the other applications) is started. To specify another locations, either set the `beastlib` environment variable to the directory (or directories where the various directories are separated by a colon, just like in Java class paths) where the jar file should be picked up.

2 Example

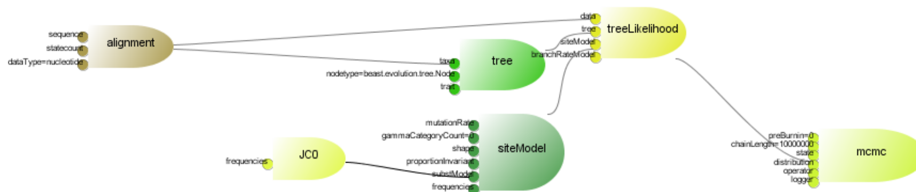


Figure 1 *Example of a model specifying Jukes Cantors substriution model (JC). It shows Plugins represented by rocket shapes connected to other Plugins through inputs (the thrusters of the rocket).*

Figure 1 show (part of) a model, representing an nucleotide sequence analysis using the Jukes Cantor substitution model. The 'rockets' represent plugins, and their thrusters the inputs. Models can be build up by connecting plugins through these inputs with other plugins. For example, in Figure 1, the Tree has an Alignment as input, and both Tree and Alignment are inputs to the TreeLikelihood. The TreeLikelihood calculates the likelihood of the sequence for a given tree. To do this, the TreeLikelihood also needs at least a SiteModel as input, and potentially also a BranchRateModel (not necessary in this example). The SiteModel specifies everything related to the transition probabilities for a site from one node to another in the Tree, such as the number of gamma categories, proportion of invariant sites and substitution model. In Figure 1, Jukes Cantor substitution model is used. In this Section, we extend this with the HKY substitution model and show how this model interacts with the operators, state, loggers and other bits and pieces in the model.

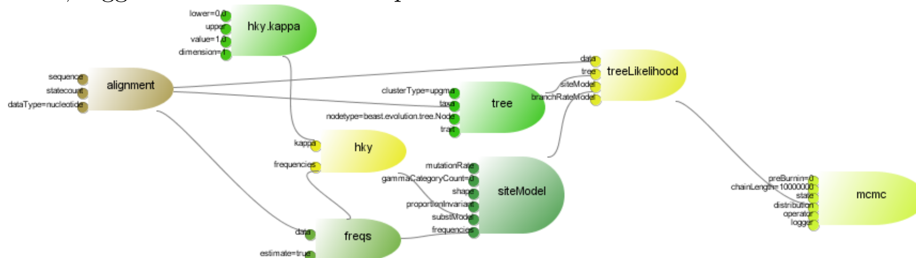


Figure 2 *Example of a model specifying a HKY substitution model.*

To define the HKY substitution model, first we need to find out what its inputs should be. The kappa parameter of the HKY model represents a variable that can be estimated. Plugins in the calculation model (i.e. the part of the model that performs the posterior calculation) are divided in StateNodes and CalculationNodes. StateNodes are classes an operator can change, while CalculationNodes are classes that change the internal state based on Inputs. The

HKY model is a CalculationNode, since it internally stores an eigenvalue matrix that is calculated based on kappa. Kappa can be changed by an operator and does not calculate antying itself, so the kappa parameter is a StateNode.
The other bit of information required for the HKY model it the character frequencies. These can be calculated from the alignment.Compare Figure 2 with Figure 1 to see how the HKY model differs from the JC model. See Section 3 for implementation details for plugin classes like HKY.

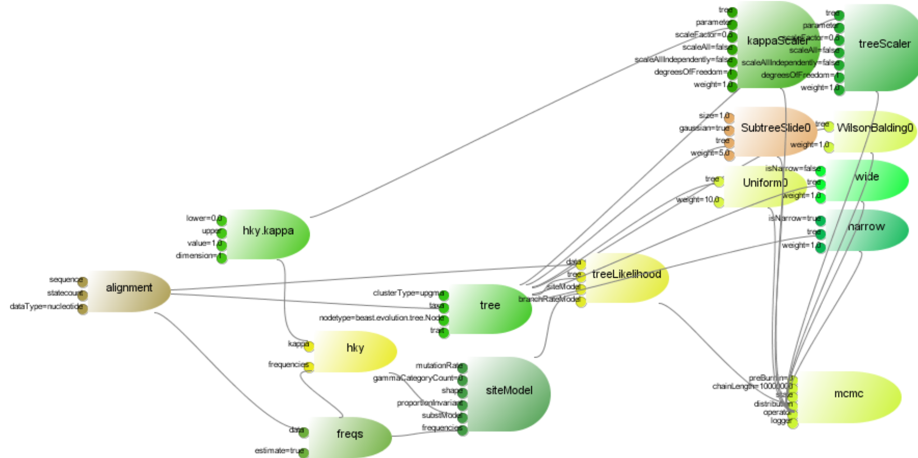


Figure 3 *Adding operators.*

In an MCMC framework, operators propose a move in the state space, and these are then accepted or rejected based on how good the moves are and luck. Figure 3 shows the HKY model extended with seven operators: six for changing the tree and one for changing the kappa parameter.

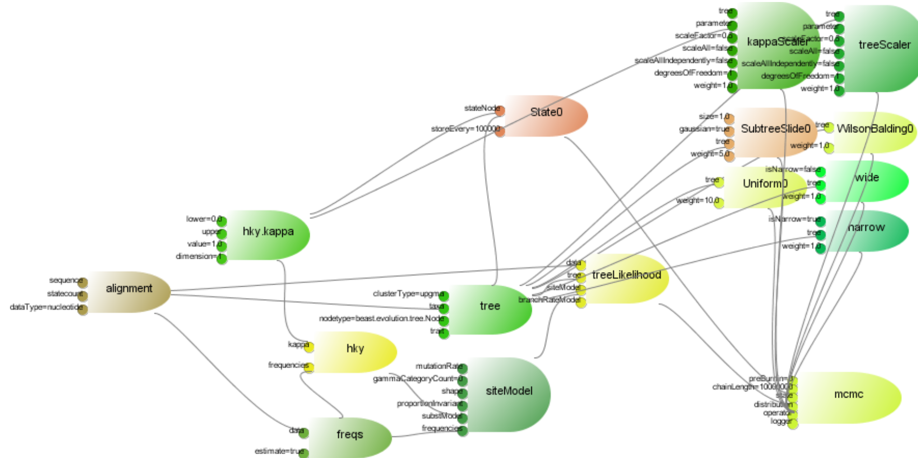


Figure 4 *Adding the state.*

The operators work on the tree and the kappa parameter. Any StateNode that an operator can work on must be part of the State. Apart from the State being a collection of StateNodes, the State performs introspection on the model and controls the order in which Plugins are notified of changes and which of them should store or restore their internal state. For example, if an operator changes the Tree, the HKY model does not need to be bothered with updating its internal state or storing that internal state since it never needs to be restored based on the tree change alone.

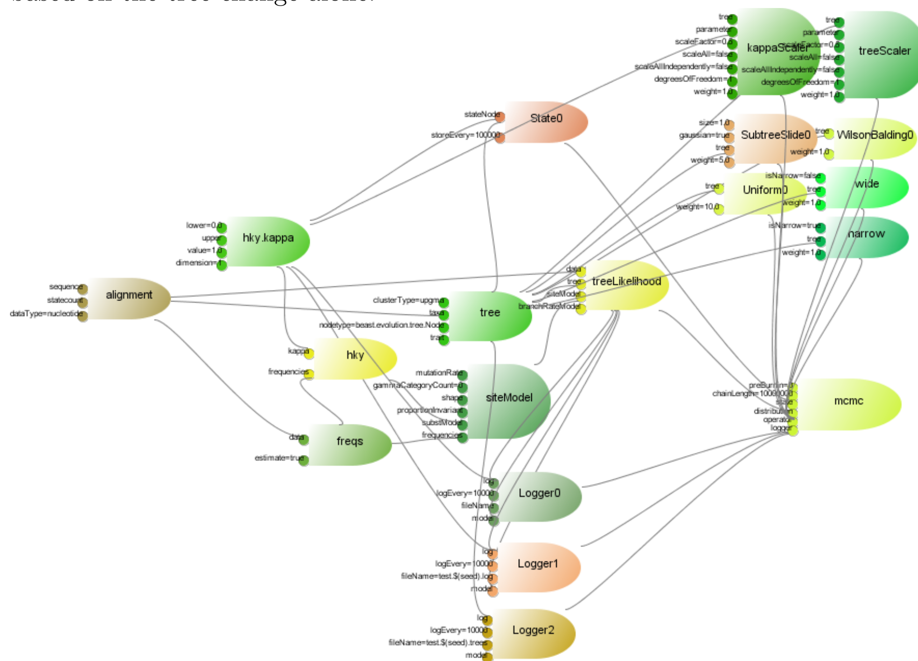


Figure 5 *Adding the loggers.*

For the MCMC analysis to be any useful, we need to log results. Loggers take care of this task. Loggers can log anything that is Loggable, such as parameters and trees, but it is easy enough to write a custom logger and add it to the list of inputs of a Logger. Typically, one logger logs to standard output, one to a log file with parameter values (a tab delimited file that can be analysed with Tracer) and one log file with trees in Newick format.

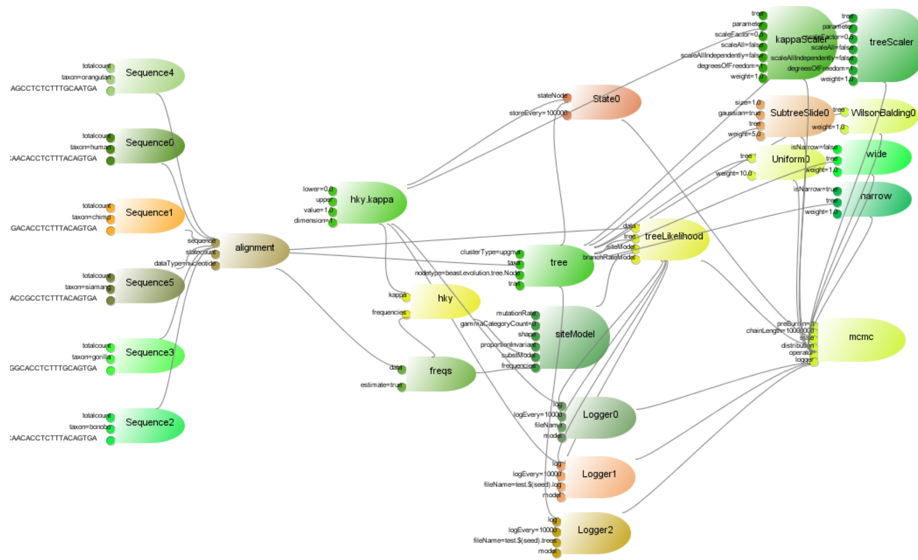


Figure 6 *Adding the sequences. This is a complete model description that can be executed in Beast II.*

Finally, the alignment consist of a list of sequences. Each sequence object containing the actual sequence and taxon information. This completes the model, shown in Figure 6 and this model can be executed by Beast II.

3 Getting started

3.1 Beast II Philosophy

Everything is a plug-in!
Plug-ins provide...

- connection with with other plug-ins/values through 'inputs'
- validation
- documentation
- 'XML parsing'

The task of a Plugin writer is to create classes, specify inputs and provide extra validation that is not already provided by inputs. The following snippet shows a very basic example.

```
@Description("Description of MyPlugin goes here")
public class MyPlugin extends Plugin {
    public Input<Integer> m_value = new Input<Integer>("value",
        "value used by my plugin");
```

```
    public void initAndValidate() throws Exception {
        // go check stuff and
        // do stuff that normally goes in a constructor
    }

} // class MyPlugin
```

Firstly, the `Description` annotation is used to provide help, which is used in GUIs online help and documentation generation. There is also a `Citation` annotation that can be used to list a reference and DOI of a publication that should be referenced when using the Plugin.
Secondly, all custom Plugins derive from Plugin or any of derived classes from Plugin. By deriving from Plugin, services through introspection like validation of models are provided.
To specify an input for a plugin, just declare an `Input` member. Input is a template class, so the type of input can be specified to make sure that when Inputs are connected to Plugins the correct type of Plugin is used. At least two strings are used in the constructor of an Input:

- a name of the input, used in the XML, in documentation and in GUIs,
- a description of the input, used in documentation and GUI help.

Other constructors exists to support validation, default values, lists of values, enumerations of Strings, etc. See Section 3.2 for details.
Finally, there is the `initAndValidate` method. This serves as a place to perform validation on the Inputs, for instance range checks or check that dimensions of two inputs are compatible. Furthermore, it is a place to perform everything that normally goes into a constructor. Plugins are typically created by the XMLParser, which firsts sets values for all inputs, then calls initAndValidate.
The following shows the skeleton of a bit larger example:

```
@Description("HKY85 (Hasegawa, Kishino & Yano, 1985) "+
        "substitution model of nucleotide evolution.")
@Citation("Hasegawa, M., Kishino, H and Yano, T. 1985. "+
        "Dating the human-ape splitting by a "+
        "molecular clock of mitochondrial DNA. " +
        "Journal of Molecular Evolution 22:160-174.")
public final class HKY extends SubstitutionModel.Base {
    public Input<RealParameter> kappa = new Input<RealParameter>("kappa",
        "kappa parameter in HKY model", Validate.REQUIRED);

    @Override
    public void initAndValidate() throws Exception {
    }

    @Override
```

```
    public void getTransitionProbabilities(double distance, double[] matrix) {
        ...
    }

    @Override
    protected boolean requiresRecalculation() {
        ...
    }

    @Override
    protected void store() {
        ...
    }

    @Override
    protected void restore() {
        ...
    }
}
```

3.2 Inputs

Inputs can be created that are primitives, plugins, lists or enumerations. By calling the appropriate constructor, the XMLParser validates the input after assigning values and can check whether a REQUIRED input is assigned a value, or whether two inputs that are XOR have exactly one input specified.

3.2.1 Input creation

Inputs can be simple primitives, like Double, Integer, Boolean, String.

```
public Input<Boolean> m_pScaleAll =
    new Input<Boolean>("scaleAll",
        "if true, all elements of a parameter (not tree) are scaled, otherwise o
        new Boolean(false));
```

Inputs of a plugin can be other plugins.

```
public Input<Frequencies> m_freqs =
    new Input<Frequencies>("frequencies",
        "frequencies nucleotide letters");
```

Inputs can be multiple inputs. When a list of inputs is specified, the Input constructor should contain a (typically empty) List as a start value.

```
public Input<List<RealParameter>> m_pParameters =
    new Input<List<RealParameter>>("parameter",
        "parameter, part of the state",
        new ArrayList<RealParameter>());
```

Inputs cannot be template classes, so `Input<Parameter<T>>` would lead to trouble. This is due to a limitation in Java introspection.
To provide an enumeration as input, the following constructor can be used: it takes the usual name and description arguments, then the default value and an array of strings to choose from. During validation it is checked that the value assigned is in the list.

```
final static String [] UNITS = {"year", "month", "day"};

public Input<String> m_sUnits = new Input<String>("units",
    "name of the units in which values are posed, " +
             "used for conversion to a real value. This can be " +
    Arrays.toString(UNITS) + " (default 'year')",
    "year",
    UNITS);
```

3.2.2 Input validation

To provide some basic validation, an extra argument can be provided to the Input constructor. By default, inputs are considered to be OPTIONAL, i.e., need not necessarily be specified. If input is REQUIRED:

```
public Input<Parameter> m_kappa =
    new Input<Parameter>("kappa",
        "kappa parameter in HKY model",
        Validate.REQUIRED);
```

If a list of inputs need to have at least one element specified, the required argument needs to be provided.

```
public Input<List<Operator>> m_operators =
    new Input<List<Operator>>("operator",
        "operator for generating proposals in MCMC state space",
        new ArrayList<Operator>(), Validate.REQUIRED);
```

Sometimes either one or another input is required, but not bot. In that case an input is declared XOR and the *other* input is provided as extra argument. The XOR goes on the second Input.

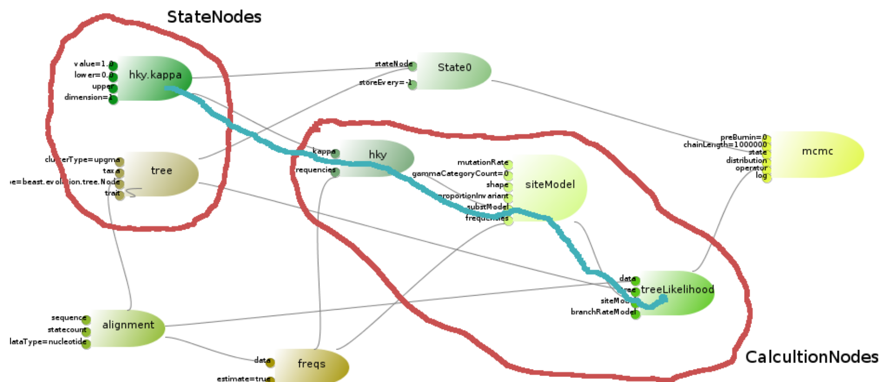
```
public Input<Tree> m_pTree =
    new Input<Tree>("tree",
        "if specified, all tree branch length are scaled");
public Input<Parameter> m_pParameter =
    new Input<Parameter>("parameter",
        "if specified, this parameter is scaled"
        , Validate.XOR, m_pTree);
```

3.3 MCMC library

The basic setup: The `MCMC` algorithm has a `State` object. `Operator` objects do proposals to change the State. A `Distribution` object calculates the posterior of the new state, and compares it with the old state's posterior (taking the Hasting ration of the `Operator` in account) to decide whether to accept or reject the new state.
Typically, a plugin developer will create new `CalculationNode`s and `Operator`s, explained below.

3.3.1 CalculationNodes

A `State` contains `StateNodes`, like `Parameter`s and `Tree`s. The `State` is responsible for calling `requiresRecalculation/store/restore` on calculation nodes. It keeps track which part of the State is changed by an Operation, hence which `CalculationNode`s may need updating.
For example, suppose a scale operator (called kappScaler) changes the kappa parameter in a simple HKY model. Then, frequencies, and the tree are not affected, but the substitution model, the site model and the tree likelihood need to be updated.



After an `Operator` has done a proposal, the `State` calls `store` on all `CalculationNodes` that could possibly be affected. Then State calls `requiresRecalculation` on each of the `CalculationNodes` between the `StateNodes` that are changed and the `Distribution` of the `MCMC`. The `requiresRecalculation` method returns true to indicate whether the CalculationNode is changed, hence is 'dirty', due to changes in the State. Also, it is a good place to set flags whether parts need to be recalculated, e.g. the EigenDecomposition in the HKY model after the kappa parameter is changed. Then, the distribution is asked to calculated the log-posterior. If the new state is accepted, `accept` is called on all `CalculationNode`s, which by default just marks them as 'not dirty'. If rejected, a `restore` is called on all `CalculationNode`s and this method can be overridden to
In summary: typically a Plugin developer overrides a `CalculationNode`.

`CalculationNode`s can help increase efficiency by overwriting

- `requiresRecalculation` to set flags to recalculate parts and return whether the Plugin is dirty or not.
- `store` to store internal states.
- `restore` to restore when a new State is not accepted.

3.3.2 Operators

`Operator`s need to have at least one `StateNode` as input. `StateNode`s are managed by the `State`, which takes care of synchronization, store and restore.
To create a new Operator, implement the `public double proposal()` method, which changes one of more `StateNode`s and returns the Hasting ratio of the proposal.

3.3.3 Loggable

To create custom loggers, implement the `Loggable` interface, which has three methods:
o init(), for generating header information and called only at the start of the log,
o exit(), for any closing statements, e.g. 'End;' in a Nexus tree file, and
o log(), for periodically logging relevant information.

3.4 Evolution library

The Evolution library provides support for calculating posteriors for phylogenetic analysis. The classes of interest are

- *SubstitutionModel* Specifies transition probability matrix for a given distance.
- *BranchRateModel* Defines a mean rate for each branch in the beast.tree.
- *SpeciationLikelihood* A likelihood function for speciation processes.
- *tree.Node* Nodes in building binary beast.tree data structure.
- *PopulationFunction* A population size function for the Coalescent.

See javadocs for details.

4 XML format

The easiest way to create XML for a newly created Plugin is to start the ModelBuilder (`java -cp beast.jar beast.app.ModelBuilder`), load an existing

XML file from the example directory and change the Plugins, then save the XML.
The basic XML is very very simple: everything can be specified using the input element. There are 4 reserved attributes, namely id, idref, name and spec.

```
<input id='myId'
     idref='otherId'
     name='inputName'
     spec='x.y.z.MyClass'  />
```

The id attribute allows elements to be referred to from other elements through the idref attribute. The name attribute specified the name of the Input in the plugin. The spec attribute specifies the Plugin class. The XML parser creates an object of this class, then set the input with name name of the Plugin specified by the enclosing input element. This way every model can be specified, but it is very tedious format to read. So, there are a lot of short cuts, making the XML more palatable.
A hand crafted XML file can be processed through the XMLParser as follows: java -cp beast.jar beast.util.XMLParser <file.xml> which then tries to beautify the XML and print it to standard output.

4.1 Short XML spec

The following elements are reserved keywords: distribution, operator, logger, data, sequence, state, parameter, tree, and run which have default mappings to objects. Furthermore, <plate var='n' range='.p1,.p2,.p3'><parameter idref='hky$(n)'/></plate> is short for <parameter idref='hky.p1'/> <parameter idref='hky.p2'/> <parameter idref='hky.p3'/> and the top level element should have attribute version='2.0' and can have namespace='x.y.z:' which allows spec-attributes to use x.y.z as name space. Finally, <map name='elementName'>x.y.z.Class</map> maps element elementName to spec x.y.z.Class.
Common abbreviations:
Element name = name attribute.

```
<input name='x'>...</input> == <x> ...  </x>
```

Primitive inputs (Integer, Double, Boolean, String) can go inline.

```
<input ...> <input name='xyz' value='1.0'/></input>
==
<input ...  xyz='1.0'/>
```

Any plugin with String constructor, like parameters and tree, can go inline

```
<input ...> <xyz spec='IntegerParameter' value='10 20'/></input>
==
<input ...  xyz='10 20'/>
```

Idref inline using @ sign.

```
<input ...> <input name='xyz' idref='ref'/></input>
==
<input ...  xyz='@ref'/>
```

5 FAQ/Known ways to get into trouble

5.1 General programming issues

5.1.1 `Input` is not declared public.

If `Inputs` are not public, they cannot get values assigned by for instance the `XMLParser`.

5.1.2 Type of input is a template class (other than `List`).

Thanks to limitations of Java introspection and the way Beast II is set up, Inputs should be of a type that is concrete, and apart from `List<T>` no template class should be used.

5.1.3 Store/restore do not call `super.store()`/`super.restore()`.

Obviously, not calling store/restore on super classes may result in unexpected behavior.

5.1.4 Input rule of base class is not what you want.

If an Input is REQUIRED for a base class you want to override, but for the derived class this Input should be OPTIONAL, set the Input to OPTIONAL in the constructor. E.g. for a SNPSequence that derives from Sequence, but for which m_sData is optional, add a constructor

```
public SNPSequence() {
        m_sData.setRule(Validate.OPTIONAL);
}
```

Note that the constructor needs to be public, to prevent IllegalAccessExceptions on construction by e.g. the XMLParser.

5.1.5 Log header on screen seems to have rubbish in there.

Put the logger that outputs to screen (i.e. does not have a filename specified) as last in the list of loggers.

5.2 Setting up

5.2.1 Setting up an add-on in Intellij

This assumes that Beast2 source code is checked out from google code using 'svn checkout http://beast2.googlecode.com/svn/trunk/ beast2-read-only' and set up as a project in Intellij, named beast2.
To set up a new Add-on to Beast 2 in Intellij

- Create new project

- Import Beast 2 as follows:
  Choose menu File/New Module
  then choose 'Import existing module'
  select the button for browsing the file to be imported.
  Then go to the Beast 2 source folder where you select Beast2.iml file.
  Press OK

- Add dependency on Beast 2 as follows:
  Choose menu File/Project structure.
  Select Modules tab
  Select your new module
  Press Add button, select Module Dependencies
  Select the Beast 2 module
  Press OK

5.2.2 Setting up an add-on in Eclipse

This assumes that Beast2 source code is checked out from google code using 'svn checkout http://beast2.googlecode.com/svn/trunk/ beast2-read-only' and set up as a project in Eclipse, named beast2.
To set up a new Add-on to Beast 2 in Eclipse

- Import beast2 as a project as follows
  Choose File/Import menu
  The 'import' dialog pops up. Choose 'General/Existing Projects into Workspace' and click 'Next'
  The next 'import' dialog pops up. Select Browse button
  Select the directory containing beast2 in the file choose dialog, click ok
  Click next, and beast2 will be added as a project

- Create new add-on project as follows
  Select File/New/Java Project menu
  Fill in project name and click the Next button
  A 'New Java Project' dialog pops up, select the 'Projects' tab
  Click 'Add' button, and a "Required Project Selection" dialog pops up
  Select 'beast2' and click OK
  Click 'Finish' and all is done

5.2.3 Set up Hudson for a google-code project add-on

This assumes that there is a build.xml file based on the build file for the beastii add-on. Hudson can be set up for a new add-on using the following steps:
1. create new job on Hudson Main Page
fill in job name
select 'Build a free-style software project'
click OK

2. Select sensible directory
Under 'Advanced Project Options' click 'Advanced' button
Click 'Use custom workspace'
In Directory entry that now appears, fill in the project name
3. Under 'Source Code Management', select 'Subversion'
Fill in repository URL, say 'http://myaddon.googlecode.com/svn/trunk'
Put a full stop in 'Local module directory (optional)'
4. set up build triggers
Click 'Build after other projects are built'
Fill in 'BEAST_2_Trunk' under 'Projects names'
Click 'Poll SCM'
Fill in '* * * * *' under 'Schedule'
5 Add build step
Click "Add build step" button, select 'Invoke Ant'
Fill in the target, e.g., build_jar_all_BEAST
6 Add Post-build actions
Click 'Publish JUnit test result report'
Fill in 'Test report XMLs' with 'build/junitreport/*.xml'
Click 'E-mail notification'
Fill in recipients
7 Click 'Save'.