General Developer Tutorial for BRAPH 2

The BRAPH 2 Developers

August 10, 2024

The software architecture of BRAPH 2 provides a clear structure for developers to understand and extend its functionalities. All objects (*elements*) in BRAPH 2 are derived from the base object `Element`. Developers can easily add new elements by writing the new elements in the simplified BRAPH 2 pseudocode. By recompiling BRAPH 2, the new elements and their functionalities are immediately integrated, also into the graphical user interface. In this developer tutorial, you will learn how BRAPH 2 is compiled, how the elements are strcutured, and how new elements can be implemented.

Contents

Compilation and Element (Re)Generation

BRAPH 2 is a compiled object-oriented programming software. Its objects are *elements*, which contain a set of *props* of various *categories* and *formats*, as described in detail in the following sections. These elements are written in the BRAPH 2 pseudocode, which simplifies and streamlines the coding process. To convert them into usable MatLab objects, BRAPH 2 needs to be compiled, which is done by calling the script `braph2genesis`, which will compile the whole BRAPH 2 code base, as shown in Code 1.

Code 1: **Compilation of BRAPH 2.** Executing the script `braph2genesis` compiles BRAPH 2 and , subsequently, unit tests it. Importantly, this script might take several hours to run (plus several more hours to unit test the compiled code).

```
>> braph2genesis
```

During the compilation, there are several phases to improve the computational efficiency of the executable code:

1. **First compilation**, where the elements are created.
2. **Second compilation**, where the elements are computationally optimized.
3. **Constant hard-coding**, where several contants are hard-coded in the executable code to further optimize the run time.

Because of this multi-stage compilation, it is not always possible to regenerate a single element without regenerating the whole BRAPH 2. Nevertheless, it is usually possible to regenerate a single element as long as the element already exists and its props have not been changed. This can be done with the function `regenerate()`, as shown in Code 2.

Code 2: **Regeneration of elements.** The function `regenerate()` can be used to regenerate some elements, as long as they already exist in the current BRAPH 2 compilation and their list of props has not been altered (e.g., renamed, moved, added), in which case it is necessary to recompile BRAPH 2 with `braph2genesis`.

```
>> close all; delete(findall(0, 'type', 'figure')); clear all (1)

>> regenerate('/src/gui', {'Pipeline'}) (2)

>> regenerate('/src/gui', {'Pipeline'}, 'DoubleCompilation', false) (3)

>> regenerate('/src/gui', {'Pipeline'}, 'CreateElement', false) (4)

```

(1) clears the workspace (not always necessary, but needed if some element instances are still in the workspace).

(2) regenerates `Pipeline`.

(3) performs only one compilation.

(4) does not regenerate the element, but only the layout and the unit test.

```
>> regenerate('/src/gui', {'Pipeline'}, 'CreateLayout', false) (5)

>> regenerate('/src/gui', {'Pipeline'}, 'CreateTest', false) (6)

>> regenerate('/src/gui', {'Pipeline'}, 'UnitTest', false) (7)

>> regenerate('/src/gui', {'Pipeline'}, 'CreateLayout', false, 'UnitTest',
    false) (8)

>> regenerate('/src/gui', {'Pipeline', 'GUI'}) (9)
```

(5) does not regenerate the layout.

(6) does not regenerate the unit test.

(7) does not perform the unit test.

(8) Multiple options can be selected at once. In this case, it does not regenerate the layout and it does not perform the unit test.

(9) Multiple elements can be regenerated at once. This can throw an error, typically because an instance of the element to be regenerated remains in the workspace. In this case, regenerate the elements one by one.

Elements

The base class for all elements is `Element`. Each element is essentially a container for a series of *props* (properties). Each prop is characterized by the following static features (i.e., equal for all instances of the prop):

- A *sequential number* (integer starting from 1).
- A *tag* (a string).
- A *category*, which determines for how a prop is used.[1]
- A *format*, which determines what a prop can contain.

[1] The possible categories and formats are shown in the boxes below.

The functions to inspect these features can be found by using the command `help Element` in the MatLab command line.

Furthermore, each instance of a prop has the following features:

- A *value*.[2] The functions to set, get, and memorize a value will be discussed in the following sections.
- A *seed* for the random number generator to ensure the reproducibility of the results. The seed of each property is a 32-bit unsigned integer and is initialized when an element is constructed by calling `randi(intmax('uint32'))`.

  The seed can be obtained using:
  `seed = el.getPropSeed(pointer)`
  where `pointer` can be either a prop number or tag. It cannot be changed.
- A *checked* status, which is true by default. Checked props are checked for format when they are set and for value when they are set/calculated.[3]

  The checked status of a prop can be altered with the functions:
  `el.checked(pointer)`

[2] The value is by defalut a `NoValue`. For `PARAMETER`, `DATA`, `FIGURE`, and `GUI` props, it can also be a callback. For `CONSTANT` props, it is usually a concrete value.

[3] When `BRAPH2.CHECKED = false`, no checks are performed. This needs to be changed in the file `BRAPH2.m`.

`el.unchecked(pointer)`
The checked status of a prop can be assessed with the function:
`checked = el.isChecked(pointer)`
where `pointer` can be either a prop number or tag.

- A *locked* status, which is false by default.[4]

  A prop can be locked with the function:
  `el.lock(pointer)`
  Once locked, it cannot be unlocked. The locked status of a prop can be assessed with the function:
  `locked = el.isLocked(pointer)`
  where `pointer` can be either a prop number or tag.

[4] The `PARAMETER` and `DATA` props get locked the first time a `RESULT` property is successfully calculated. The locked status is not used for `CONSTANT` props.

- A *callback* instance.[5]

  The callback to a prop can be obtained using the function:
  `cb = el.getCallback(pointer)`
  where `pointer` can be either a prop number or tag.

[5] Callbacks are not used with `METADATA` props.

Additional functions to operate with these features can be found by using the command `help Element` in the MatLab command line.

Property Categories

`CONSTANT` Static constant equal for all instances of the element. It allows incoming callbacks.
`METADATA` Metadata NOT used in the calculation of the results. It does not allow callbacks. It is not locked when a result is calculated.
`PARAMETER` Parameter used to calculate the results of the element. It allows incoming and outgoing callbacks. It is connected with a callback when using a template. It is locked when a result is calculated.
`DATA` Data used to calculate the results of the element. It is `NoValue` when not set. It allows incoming and outgoing callbacks. It is locked when a result is calculated.
`RESULT` Result calculated by the element using parameters and data. The calculation of a result locks the element. It is `NoValue` when not calculated. It allows incoming callbacks.
`QUERY` Query result calculated by the element. The calculation of a query does NOT lock the element. It is `NoValue` when not calculated. Typically, it should not be memorized. It does not allow callbacks.
`EVANESCENT` Evanescent variable calculated at runtime (typically employed for handles of GUI components). It is `NoValue` when not calculated. Typically, it should be memorized at first use. It does not allow callbacks.
`FIGURE` Parameter used to plot the results in a figure. It allows incoming and outgoing callbacks. It is not locked when a result is calculated.
`GUI` Parameter used by the graphical user interface (GUI). It allows incoming and outgoing callbacks. It is not locked when a result is calculated.

Property Formats

`EMPTY` Empty has an empty value and is typically used as a result or query to execute some code.
`STRING` String is a char array.
`STRINGLIST` StringList is a cell array with char arrays.
`LOGICAL` Logical is a boolean value.
`OPTION` Option is a char array representing an option within a set defined in the element (case sensitive). Settings: cell array of chars representing the options, e.g., `{'plus', 'minus', 'zero'}`.
`CLASS` Class is a char array corresponding to an element class. Settings: class name of a subclass of Element (or Element itself).
`CLASSLIST` ClassList is a cell array with char arrays corresponding to element classes. Settings: class name of a subclass of Element (or Element itself), which represents the base element.
`ITEM` Item is a pointer to an element of a class defined in the element. Settings: class name of a subclass of Element (or Element itself).
`ITEMLIST` ItemList is a cell array with pointers to elements of a class defined in the element. Settings: class name of a subclass of Element (or Element itself), which represents the base element.
`IDICT` Idict is an indexed dictionary of elements of a class defined in the element. Settings: class name of a subclass of Element (or Element itself), which represents the dictionary element.
`SCALAR` Scalar is a scalar numerical value.
`RVECTOR` RVector is a numerical row vector.
`CVECTOR` CVector is a numerical column vector.
`MATRIX` Matrix is a numerical matrix.
`SMATRIX` SMatrix is a numerical square matrix.
`CELL` Cell is a 2D cell array of numeric data, typically used for adjaciency matrices and measures.
`NET` Net is a MatLab neural network object (network, SeriesNetwork, DAGNetwork, dlnetwork).
`HANDLE` Handle is a handle for a graphical or listener component. It should only be used as an evanescent property.
`HANDLELIST` HandleList is a cell array with handles for graphical or listener components. It should only be used as an evanescent property.
`COLOR` Color is an RGB color, e.g., `'[1 0 0]'` for red.
`ALPHA` Alpha is a transparency level between 0 and 1.
`SIZE` Size represents the size of a graphical component. It is a positive number (default = 1).
`MARKER` Marker represents the marker style. It can be `'o'`, `'+'`, `'*'`, `'.'`, `'x'`, `'_'`, `'|'`, `'s'`, `'d'`, `'^'`, `'v'`, `'>'`, `'<'`, `'p'`, `'h'`, `''` (no marker).
`LINE` Line represents the line style. It can be `'-'`, `':'`, `'-.'`, `'-'`, `''` (no line).

Even though it is possible to create instances of `Element`, it does not have any props and typically one uses its subclasses. Its three direct subclasses are `NoValue`, `Callback`, and `ConcreteElement`, as shown in Figure 1.

The element `NoValue` is used to represent a value that has not been set (for properties of categories `METADATA`, `PARAMETER`, `DATA`, `FIGURE` or `GUI`) or calculated (for properties of category `RESULT`, `QUERY`,

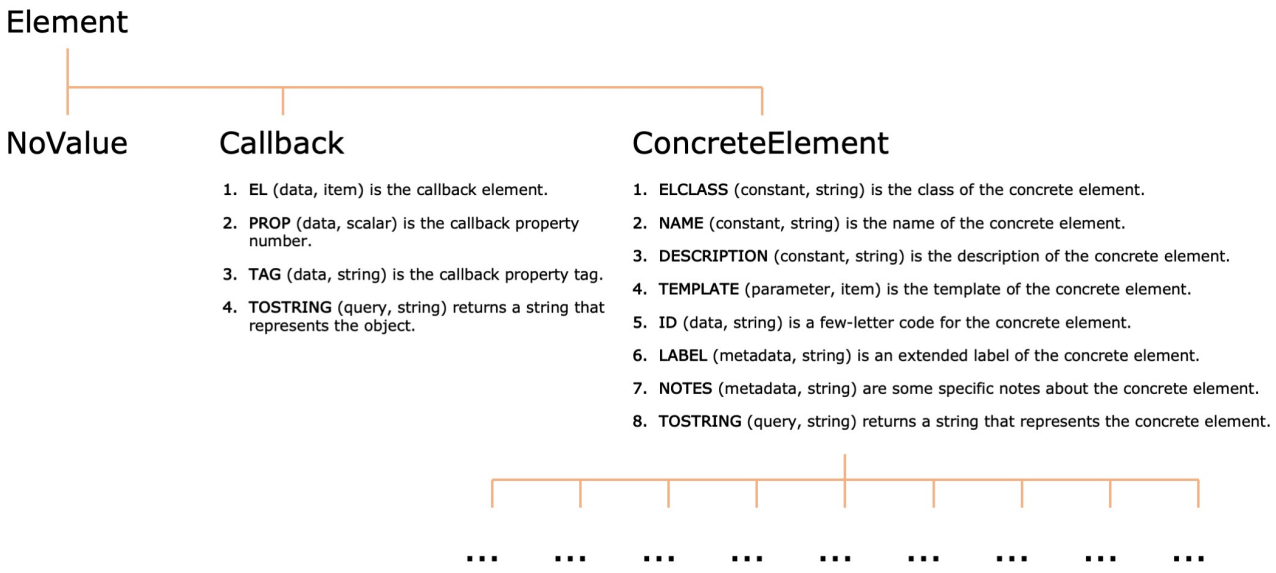

Figure 1: **Element tree.** All elements derive from the base class `Element`. Its direct children are `NoValue`, `Callback`, and `ConcreteElement`, whose properties are also indicated. Concrete elements further derive directly or indirectly from `ConcreteElement`.

`EVANESCENT`), while it should not be used for properties of category `CONSTANT`. It should be instantiated using Code 3.

Code 3: **Instantiation of `NoValue`.** For computational efficiency, it is best to use only one instance using this script, instead of creating new instances using the constructor `NoValue()`.

```
Element.getNoValue()
```

No element can be a subclass of NoValue.

A `Callback` refers to a prop of another element `el`, identified by prop number or tag. It should be instantiated using Code 4.

Code 4: **Instantiation of a `Callback`.** For computational efficiency, it is best to use only one instance of `Callback` for each prop of an instance of a concrete element `el` with the code shown below, instead of creating new callback instances using its constructor.

```
el.getCallback('PROP', PROP_NUMBER)
el.getCallback('TAG', PROP_TAG)
```

No element can be a subclass of `Callback`.

A concrete element (`ConcreteElement`) provides the infrastructure necessary for all concrete elements. In particular, it has the constant props `ELCLASS` (string), `NAME` (string) and `DESCRIPTION` (string), the property `TEMPLATE` (item), the indexing properties `ID` (string), `LABEL`

(string), and `NOTES` (string), and the query prop `TOSTRING` (string). Even though it is possible to create instances of `ConcreteElement`, typically one uses its subclasses.

Setting Props

The value of a prop can be set with Code 5.

Code 5: **Setting a prop.** This script illustrates various ways in which props can be set.

```
el.set('ID', 'new el id') (1)
el.set(5, 'new el id') (2)

el.set( ... (3)
  'ID', 'new el id', ...
  'LABEL', 'new el label', ...
  7, 'new el notes' ...
  )

el = el.set('ID', 'new el id') (4)
```

(1) and (2) set the value of a prop with the prop tag or the prop number.

(3) sets the values of multiple props at once. The pointers can be either property numbers or property tags.

(4) returns the element.

When a prop is set to a certain value, the following operations are performed:

1. The value is **conditioned** before being set (by calling the protected *static* function `conditioning()`, which can be defined in each subelement).

   This can be set with the token `!conditioning`.

2. The value is **preset** before being set (by calling the protected function `preset()`, which can be defined in each subelement).[6]

   This can be set with the token `!preset`.

3. If a property is checked, its **format is checked** before proceeding to its setting by calling `Format.checkFormat()`.
   If the check fails, the property is not set and an error is thrown with error id `BRAPH2:<Element Class>:WrongInput`.

   This can be set with the token `!checkProp`.

4. The value is **set**.

   If the property is of category `PARAMETER`, `DATA`, `FIGURE`, or `GUI`, the value is set only if the property is unlocked.
   If an attempt is made to set a locked property, no setting occurs and a warning is thrown with warning id `BRAPH2:<Element Class>`.

[6] Differently from the *static* function `conditioning()`, the function `preset()` has access to the element instance.

If the value is a callback, a warning is thrown if the element, property number and/or settings of the callback do not coincide with those of the property with warning id `BRAPH2:<Element Class>`.

If the property is of category `RESULT`, `QUERY` or `EVANESCENT`, the value can only be set to `Element.getNoValue()`.

5. The value is **postset** after being set (by calling the protected function `postset()`, which is defined in each subelement).

   This can be set with the token `¡postset!`.

6. **All props** are **postprocessed** after being set (by calling the protected function `postprocessing()`, which is defined in each subelement).

   This can be set with the token `¡postprocessing!`.

7. If ANY property is checked, the function `Element.check()` is called after all settings are made and the consistency of the values of **all pros** are **checked**.
   If the check fails an error is thrown with error id `BRAPH2:<Element Class>:WrongInput`.

8. When a prop is successfully set, an **event** `PropSet()` is **notified**.

Getting Props

The value of a prop can be retrieved with Code 6.

Code 6: **Getting a prop.** This script illustrates various ways in which the value of a prop can be retrieved.

```
value = el.get('ID'); (1)

value = el.get(ConcreteElement.ID); (2)

el.get('ID') (3)
el.get(ConcreteElement.ID) (4)

value = el.get('QUERY', ARG1, ARG2, ... ); (5)
```

(1) gets the value of a prop using the prop tag.
(2) gets the value of a prop using the prop number.
(3) and (4) do not return any output value. This can be useful, e.g., when a code needs to be executed, e.g., by a `QUERY`.
(5) can be used with a series of arguments for props of category `QUERY`. Any additional arguments are ignored for props of other categories.

If the raw value of the property is a `NoValue`, it proceed to return the default property value (for categories `METADATA`, `PARAMETER`, `DATA`, `FIGURE`, and `GUI`).

If the raw value of the property is a callback, it retrieves the value of the linked property (for categories `PARAMETER`, `DATA`, `FIGURE`, and `GUI`).

If a property of category `RESULT`, `QUERY`, or `EVANESCENT` is not calculated (i.e., its raw value is `NoValue`), it proceeds to calculate it (but not to memorize it, i.e., its raw value remains `NoValue`). After the calculation of a property of category `RESULT` all properties of categories `PARAMETER` and `DATA` are irreversibly locked. If the property is checked, it proceeds to check all properties after the calculation calling the function `check()`. If the check fails, it resets the property to `NoValue` and returns `NoValue`, does not lock the property, and throws a warning with warning id `BRAPH2:<Element Class>`.

The raw value of a prop can be retrieved with Code 7.

Code 7: **Getting the raw value of a prop.** This script illustrates various ways in which the raw value of a prop can be retrieved.

```
value = el.getr('ID');
value = el.getr(ConcreteElement.ID);
```

Memorizing Props

The value of a prop can be memorized using Code 8.

Code 8: **Memorizing a prop.** This script illustrates various ways in which the value of a prop can be memorized.

```
value = el.memorize('ID'); (1)
value = el.memorize(ConcreteElement.ID); (2)

el.memorize('ID') (3)
el.memorize(ConcreteElement.ID) (4)
```

(1) and (2) memorize the value of a prop using the prop tag and the prop number.

(3) and (4) do not return any output value.

If the property is of category `RESULT`, `QUERY`, or `EVANESCENT`, it calls the function check, proceed to save the result, and notifies an **event PropMemorized**.

If the property is *not* of category `RESULT`, `QUERY`, or `EVANESCENT` and has not been set yet, it sets it to its default value.

If the property is *not* of category `RESULT`, `QUERY`, or `EVANESCENT` and is a callback, it iteratively memorizes the property of the element in the callback.

If a property of category `QUERY` is memorized, a warning is thrown with warning id `BRAPH2:<Element Class>`, because query properties are generally not supposed to be memorized. If such behavior is intended, consider enclosing the command between warning off and warning on.

Element tokens

A generator file has the structure illustrated Code 9.

Code 9: **Element tokens in a generator file.** All tokens available in a generator file. The name of this file must end with `.gen.m`, and tipically starts with `_`. The token `¡header!` is required (and the token `¡build!`), while the rest is optional.

```
%% ¡header!
  <class_name> < <superclass_name> (<moniker>, <descriptive_name>) <header_description>.
  %%% ¡class_attributes!
   Class attributes is a single line, e.g. Abstract = true, Sealed = true.
  %%% ¡description!
   This is a plain description of the element.
   It can occupy several lines.
  %%% ¡seealso!
   Related functions and classes in a single line, coma-separated and without fullstop.
  %%% ¡build!
  Number of the build of the element starting from 1.

 %% ¡constants!
  Constants.

 %% ¡props!
  %%% ¡prop!
   <tag1> (<category>, <format>) <description>.
   %%%% ¡settings!
    Prop settings, depending on format.
   %%%% ¡default!
    Prop default value (seldom needed).
   %%%% ¡conditioning!
    Code to condition value (before checks and calculation).
    Can be on multiple lines.
    The prop value is in the variable 'value',
    where also the conditioned prop value is returned.
   %%%% ¡preset!
    Code to preset element (before checks and calculation).
    Can be on multiple lines.
    The prop value is in the variable 'value',
    where also the preset prop value is returned.
   %%%% ¡check_prop!
    Code to check prop format (before calculation).
    Can be on multiple lines.
    The prop value is in the variable 'value'.
    The outcome should be in variable 'check'.
   %%%% ¡postset!
    Postset code (executed after setting, but before checking, value),
    executed on ONLY the set property.
    Can be on multiple lines.
    Does not return anything.
   %%%% ¡postprocessing!
    Postprocessing code (executed after setting, but before checking,
    value), executed on ALL unlocked props after each set operation.
    Can be on multiple lines.
    Does not return anything.
   %%%% ¡check_value!
    Code to check prop value (after calculation).
    Can be on multiple lines.
    The prop value is in the variable 'value'.
    The outcome should be in variable 'check' and the message in 'msg'.
```

```
   %%%% icalculate!
    Code to calculate prop results (only for category RESULT).
    Can be on multiple lines.
    Can include callbacks as {@cb_get, 'TAG', varargin} and
    {@cb_set, 'TAG1', value1, ...}.
    The result should be in variable 'value'.
   %%%% icalculate_callbacks!
    Callbacks to be used in calculate, typically as functions
    cb_name(src, event).
    Can be on multiple lines.
   %%%% igui!
    GUI code for representing the panel of the prop.
    Can be on multiple lines.
    Should return a PanelProp object in 'pr'.
  %%% iprop!
    <tag2> ...

%% iprops_update!
 %%% iprop!
  <tag1> (<category>, <format>) <description>. [Only description can be different from original prop]
  %%%% isettings!
   Updated settings.
  %%%% idefault!
   Updated default.
  %%%% iconditioning!
   Update value conditioning (before checks and calculation).
  %%%% ipreset!
   Update element value preset (before checks and calculation).
  %%%% icheck_prop!
   Updated check prop format (before calculation).
  %%%% ipostset!
   Update postset (after setting, but before checking, value).
  %%%% ipostprocessing!
   Update value postprocessing (after setting, but before checking, value).
  %%%% icheck_value!
   Updated check prop value (after calculation).
  %%%% icalculate!
   Updated calculation.
  %%%% icalculate_callbacks!
   Updated calculate callbacks.
  %%%% igui!
   Updated GUI.
 %%% iprop!
  <tag2> ...

%% igui!
 %%% imenu_import!
  Menu Import for the GUI figure.
  The element is el.
  The menu is menu_import.
  The plot element is pe.
 %%% imenu_export!
  Menu Export for the GUI figure.
  The element is el.
  The menu is menu_export.
  The plot element is pe.

 %% ilayout!
 %%% iprop!
  %%% iid!
   Prop id, e.g., Element.TAG, ordered as they should appead.
```

```
114    %%% !title!
115     String containing the title of the prop panel.
116   %%% !prop!
117    ...
118
119  %% !tests!
120   %%% !excluded_props!
121    Row vector with list of props to be excluded from standard tests.
122   %%% !warning_off!
123    Switches off the warnings regarding the element.
124   %%% !test!
125    %%%% !name!
126    Name of the text on a single line.
127    %%%% !probability!
128    Probability with which this test is performed. By default it is 1.
129    %%%% !code!
130    Code of the test.
131    Can be on multiple lines.
132   %%% !test!
133    ...
134   %%% !test_functions!
135   Functions used in the test.
136   Can be on multiple lines.
```

A list of special instructions is shown in Code 10.

Code 10: **Special instruction in a generator file.** There are some special and specialized instructions that can be used in a generator file.

```
1  €ConcreteElement.NAME€ (1)
2
3  __Category.CONSTANT__ (2)
4  __Category.CONSTANT_TAG__ (3)
5  ...
6  __Format.EMPTY__ (4)
7  __Format.EMPTY_TAG__ (5)
8  ...
9
10 %%%__WARN_TBI__ (6)
```

(1) substitutes the prop with its default value, when hard-coding the element. (2) keeps `Category.CONSTANT` even after hard-coding the element, instead of substituting it with its value. (3) — (5) It works similarly also for the other constants of `Category` and `Format`.

(6) adds a warning that the specific feature is not implemented yet.

Overview of Elements

The directory structure of `braph2` and the relation with `braph2genesis` is illustrated in Figure 2. All objects are derived from a base object called `Element` and written in a simplified pseudocode (files `*.gen.m`) that is compiled into the actual elements (files `*.m`) by the command `braph2genesis` (some examples of these elements are shown). The compiled code can be launched by the command `braph2`. The core of BRAPH 2 (gray shaded area) includes the compiler (`genesis`), the essential source code (`src`), and the essential functionalities for the GUI (`gui`, yellow-shaded area). The users can easily add new

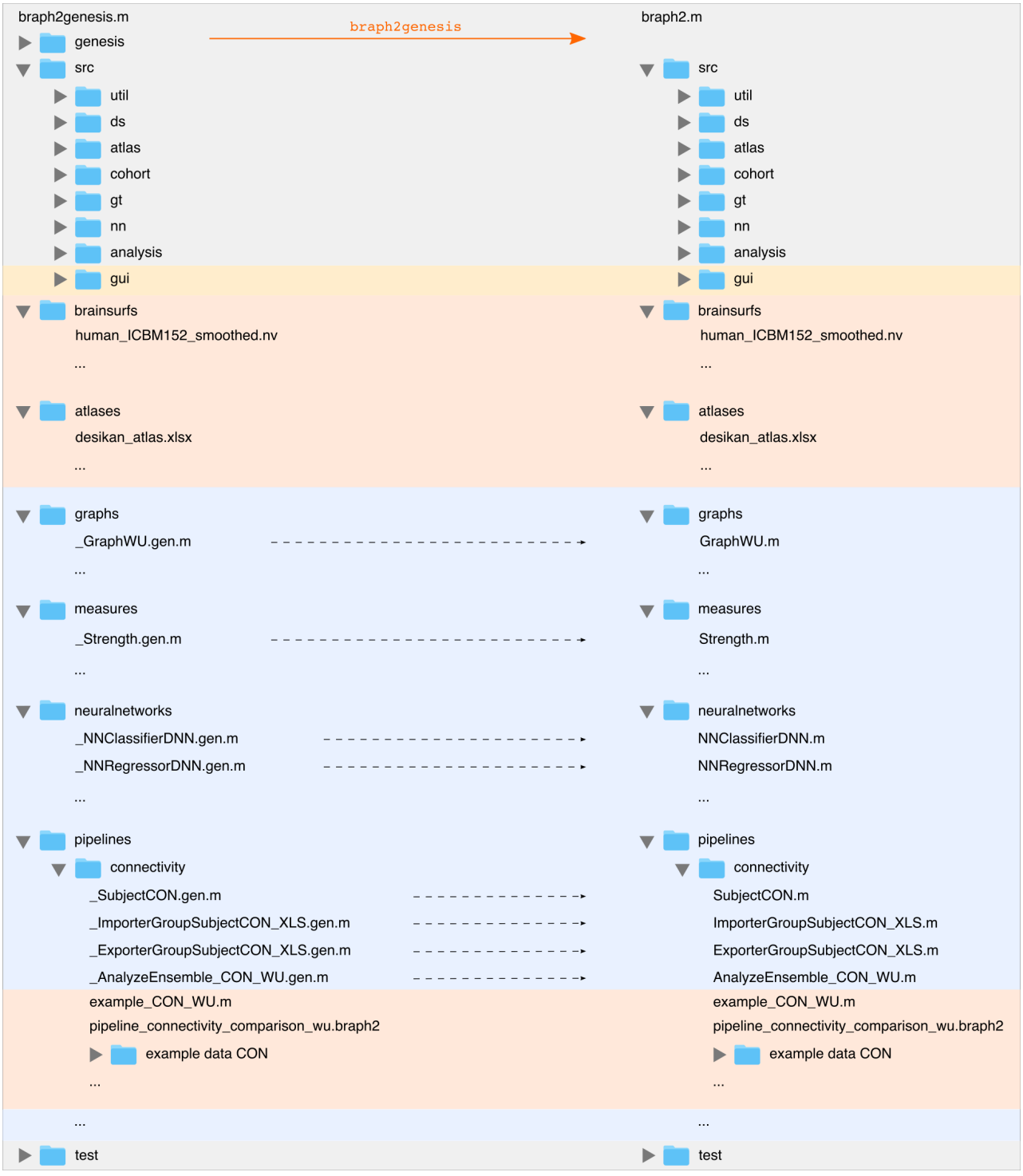

Figure 2: **BRAPH 2 genesis..** Directory structure of `braph2genesis` (left) and `braph2` (right).

brain surfaces (`brainsurfs`), atlases (`atlases`), example scripts and GUI pipelines (in the corresponding folder under `pipelines`). Furthermore, the users can add new elements such as new graphs (e.g., `GraphWU` in `graphs`), measures (e.g., `Strength` in `measures`), data types (e.g., `SubjectCON` in `pipelines/connectivity`), data importers (e.g., `ImporterGroupSubjectCON_XLS` in `pipelines/connectivity`), data exporters (e.g., `ExporterGroupSubjectCON_XLS` in `pipelines/connectivity`), and analyses (e.g., `AnalyzeEnsemble_CON_WU` in `pipelines/connectivity`) by writing new elements and recompiling the whole code: the new elements and their functionalities will be immediately available also in the GUI. Finally, BRAPH 2 is provided with a set of unit tests (executable by the command `test_braph2`) that ensure the formal correctness of the code, including that of any newly added elements.

Implementation of an Element

We will now see how to implement a few concrete elements.

Light compilation of BRAPH 2

To speed up the compilation of BRAPH 2 when trying these examples, it is possible to perform a light version of the compilation using the script `braph2genesis_with_rollcall`, which permits one to exclude/include specific folders or elements, as shown in Code 11.

Code 11: **BRAPH 2 genesis with rollcall.** Using `braph2genesis_with_rollcall` (which is found in the folder `sandbox`), it is possible to exclude some folders and elements, which are defined in the variable `rollcall`. You can place your elements in the folder `sandbox`.

```
...
%% Add here all included and excluded folders and elements
% '-folder'                 the folder and its elements will be excluded
%
% '+folder'                 the folder is included, but not its elements
%   '+_ElementName.gen.m'   the element is included,
%                           if the folder is included
%
% '+folder*'                the folder and its elements are included
%   '-_ElementName.gen.m'   the element is excluded,
%                           if the folder and its elements are included
% (by default, the folders are included as '+folder*')
rollcall = { ...
    '+util*', '-_Exporter.gen.m', '+_Importer.gen.m', ...
    '+ds*', '-ds_examples', ...
    '-atlas', ...
    '-gt', ...
    '-cohort', ...
    '-analysis', ...
    '-nn', ...
    '-gui', '-gui_examples', ...
    '-brainsurfs', ...
    '-atlases', ...
    '-graphs', ...
    '-measures', ...
    '-neuralnetworks', ...
    '-pipelines', ...
    '+test*', ...
    '+sandbox*' ...
    };
...
```

This same approach (appropriately altering the included/excluded folders and elements) can also be used to reduce the compilation time when developing new functionalities.
Importantly, the directory `braph2genesis` must be in the MatLab file path and the directory `braph2` must not be in the MatLab file path. The compiled BRAPH 2 is saved in `brap2_with_rollcall`, which is ignored by GIT.

A Simple Calculator

You will now create your first element (Code 12), a simple calculator that contains two numbers (which are data scalar props) and calculates their sum and difference (which are result scalar props).

Code 12: **Arithmetic Operation Calculator.** This is a simple element direclty deriving from `ConcreteElement`.

```
%% iheader! (1)
ArithmeticOperations < ConcreteElement (ao, arithmetic operation calculator)
      calculates simple arithmetic operations.

%%% idescription!
An Arithmetic Operation Calculator (ArithmeticOperations) contains two
 numbers as data scalar props and calculates their sum and difference as
 result scalar props.

%%% iseealso!
LogicalOperations, GeometricalOperations

%%% ibuild! (2)
1


%% iprops_update! (3)

%%% iprop!
ELCLASS (constant, string) is the class of the arithmetic operation
     calculator.
%%%% idefault!
'ArithmeticOperations' (4)

%%% iprop!
NAME (constant, string) is the name of the arithmetic operation calculator.
%%%% idefault!
'Arithmetic Operation Calculator'

%%% iprop!
DESCRIPTION (constant, string) is the description of the arithmetic
     operation calculator.
%%%% idefault!
'An Arithmetic Operations element (ArithmeticOperations) contains two
     numbers as data scalar props and calculates their sum and difference as
      result scalar props.'

%%% iprop!
TEMPLATE (parameter, item) is the template of the arithmetic operation
     calculator.
%%%% isettings!
'ArithmeticOperations' (5)

%%% iprop!
ID (data, string) is a few-letter code for the arithmetic operation
     calculator.
%%%% idefault!
'ArithmeticOperations ID'

%%% iprop!
LABEL (metadata, string) is an extended label of the arithmetic operation
     calculator.
%%%% idefault!
'ArithmeticOperations label'

%%% iprop!
NOTES (metadata, string) are some specific notes about the arithmetic
```

(1) The `iheader!` and (2) `ibuild!` tokens are the only required one.

(3) The `iprops_update!` token permits to update the properties of the `ConcreteElement`. The updated parts have been highlighted.

(4) must be the name of the element.

(5) must be the name of the element.

```
     operation calculator.
%%%% idefault!
'ArithmeticOperations notes'

%%% iprop! (6)
TOSTRING (query, string) returns a string that represents the  arithmetic
     operation calculator.
%%%% icalculate! (7)
a = ao.get('A');
b = ao.get('B');
value = ['Calculator of the sum and difference of ' num2str(A) ' and '
     num2str(B)];


%% iprops! (8)

%%% iprop! (9)
A (data, scalar) is the first number.

%%% iprop! (10)
B (data, scalar) is the second number.

%%% iprop! (11)
SUM (result, scalar) is the sum of the two numbers (A + B).
%%%% icalculate! (12)
value = ao.get('A') + ao.get('B');

%%% iprop! (13)
DIFF (result, scalar) is the difference of the two numbers (A - B).
%%%% icalculate! (14)
value = ao.get('A') - ao.get('B');


%% itests! (15)

%%% itest!
%%%% iname!
Simple test
%%%% icode!
ao = ArithmeticOperations('A', 6, 'B', 4)

string = ao.get('TOSTRING')
assert(~ao.isLocked('A')) (16)
assert(~ao.isLocked('B')) (17)

sum = ao.get('SUM')

assert(ao.isLocked('A')) (18)
assert(ao.isLocked('B')) (19)

diff = ao.get('DIFF')

sum_raw = ao.getr('SUM') (20)
diff_raw = ao.getr('DIFF') (21)
```

(6) Often, it is not necessary to updated `TOSTRING`, as the default works for most cases.

(7) returns the string, which must be saved in the variable `value`.

(8) The `iprops!` token permits to add additional props.

(9) and (10) are two data props.

(11) is a result prop.

(12) calculates the sum of the two numbers. The result must be saved in the variable `value`.

(13) is a result prop.

(14) calculates the difference of the two numbers. The result must be saved in the variable `value`.

(15) The `itests!` token permits to add unit tests.

(16) and (17) Both props `A` and `B` are not locked, even though the query prop `TOSTRING` has been calculated.

(18) and (19) Both props `A` and `B` are now locked, because the result prop `SUM` has been calculated. From now on their value cannot be changed.

(20) and (21) Note that both the result props `SUM` and `DIFF` are `NoValue`, because they have not been memorized yet.

```
assert(isa(sum_raw, 'NoValue') && isa(diff_raw, 'NoValue'))

%%% itest!  (22)
%%%% iname!
Simple test with memorization
%%%% icode!
ao = ArithmeticOperations('A', 6, 'B', 4)

sum = ao.memorize('SUM')
diff = ao.memorize('DIFF')

sum_raw = ao.getr('SUM')
diff_raw = ao.getr('DIFF')
assert(~isa(sum_raw, 'NoValue') && ~isa(diff_raw, 'NoValue'))
```

(22) alters the previous test to memorize the results.

Calculator with Seeded Randomness

You can now create an element that demonstrate how the seeded randomness works (Code 13).

Code 13: **Arithmetic Operation Calculator.** This is a simple element directly deriving from `ConcreteElement`.

```
%% iheader!
SeededRandomness < ConcreteElement (sr, randomizer) generates a random
     number.

%%% idescription!
... (1)

%%% ibuild!
1


%% iprops_update!

%%% iprop!
ELCLASS (constant, string) is the class of the randomizer.
%%%% idefault!
'SeededRandomness'

... (2)


%% iprops!

%%% iprop!
RANDOM_NUMBER (result, scalar) is a random number.
%%%% icalculate!
value = rand();


%% itests!

%%% itest!
Simple test
%%%% icode!
sr1 = SeededRandomness()
```

(1) Here, a detailed description should be provided.

(2) Here, the other standard properties derived from `ConcreteElement` should be updated as well (with the possible exception of `TOSTRING`).

```
sr2 = SeededRandomness()

assert(sr1.get('RANDOM_NUMBER') == sr1.get('RANDOM_NUMBER')) (3)
assert(sr2.get('RANDOM_NUMBER') == sr2.get('RANDOM_NUMBER')) (5)
assert(sr1.get('RANDOM_NUMBER') ~= sr2.get('RANDOM_NUMBER')) (6)
```

(3) and (4) check that subsequent calls to the calculation of the random number return the same value.
(6) checks that calls to the calculation of the random number of differen randomizers return different values.

Query

You can now learn how to use query props by expanding the `ArithmeticOperations` (Code 14).

Code 14: **Arithmetic Operation Calculator with Queries.** This element derives from `ArithmeticOperations` to include a query with arguments.

```
%% iheader!
ArithmeticOperationsWithQuery < ArithmeticOperations (ao, calculator with
     query) calculates simple arithmetic operations with a query.

%%% idescription!
...

%%% ibuild!
1


%% iprops_update!

%%% iprop!
ELCLASS (constant, string) is the class of the calculator with query.
%%%% idefault!
'ArithmeticOperationsWithQuery'

...


%% iprops!

%%% iprop!
SUM_OR_DIFF (query, scalar) returns the sum or difference depending on the
     argument.
%%%% icalculate!
% R = ao.get('SUM_OR_DIFF', SUM_OR_DIFF) returns the sum of A and B if (1)
%  SUM_OR_DIFF = 'SUM' or the difference of A and B if SUM_OR_DIFF = 'DIFF'.

if isempty(varargin) (2)
    value = NaN;
    return
end
sum_or_diff = varargin{1};

switch sum_or_diff
    case 'SUM'
        value = ao.get('SUM');

    case 'DIFF'
        value = ao.get('DIFF');
```

(1) It is good practice to add some comments about the arguments for the query.
(2) It is also good practice to check the input arguments and provide a reasonable output for absent/unexpected arguments.

```
41
42      otherwise
43          value = NaN;
44  end
45
46
47
48  %% itests!
49
50  %%% itest!
51  Simple test
52  %%%% icode!
53  ao = ArithmeticOperationsWithQuery('A', 6, 'B', 4)
54
55  assert(ao.get('SUM_OR_DIFF', 'SUM') == ao.get('SUM')) (3)
56  assert(ao.get('SUM_OR_DIFF', 'DIFF') == ao.get('DIFF')) (4)
57  assert(isnan(ao.get('SUM_OR_DIFF'))) (5)
58  assert(isnan(ao.get('SUM_OR_DIFF', 'anything else'))) (6)
```

(3) and (4) return the sum or the difference depening on the argument. (5) and (6) return `NaN` when the input is absent or unexpected.

Evanescent, Gui, Figure

You can now learn how to use evanescent props and graphical handles (Code 15).

Code 15: **Element with figure.** Element with a figure to illustrate how to use evanescent handles.

```
1   %% iheader!
2   ElementWithFigure < ConcreteElement (ef, element with figure) is an element
        with a figure.
3
4   %%% idescription!
5   ...
6
7   %%% ibuild!
8   1
9
10
11  %% iprops_update!
12
13  %%% iprop!
14  ELCLASS (constant, string) is the class of the element with figure.
15  %%%% idefault!
16  'ElementWithFigure'
17
18  ...
19
20
21  %% iprops!
22
23  %%% iprop!
24  FIG (evanescent, handle) is the handle of a figure.
25  %%%% icalculate!
26  value = uifigure( ... (1)
27      'Name', 'Figure from ElementWithFigure', ...
28      'Color', BRAPH2.COL ...
29      );
30
```

(1) renders a figure and returns its handle.

```
%%% iprop!
PANEL (evanescent, handle) is the handle of the panel.
%%%% icalculate!
if ~check_graphics(ef.memorize('FIG'), 'figure') (2)
    ef.set('FIG', Element.getNoValue()); (3)
end

fig = ef.memorize('FIG'); (4)

value = uipanel( ...
    'Parent', fig, ... (5)
    'Units', 'normalized', ...
    'Position', [.25 .25 .50 .50], ...
    'BackgroundColor', BRAPH2.COL_BKG ...
    );

%%% iprop!
BUTTONS (evanescent, handlelist) is the list of handles of the buttons.
%%%% icalculate!
if ~check_graphics(ef.getr('PANEL'), 'uipanel') (6)
    ef.set('PANEL', Element.getNoValue()); (7)
end

panel = ef.memorize('PANEL'); (8)

value = {};
for i = 1:1:10
    value{i} = uibutton( ...
        'Parent', panel, ... (9)
        'Text', ['B' int2str(i)], ...
        'Position', [ ...
            (i - 1) * w(panel, 'pixels') / 10 ...
            (i - 1) * h(panel, 'pixels') / 10 ...
            w(panel, 'pixels') / 10 ...
            h(panel, 'pixels') / 10 ...
            ], ...
        'ButtonPushedFcn', {@cb_button} ... (10)
        );
end
%%%% icalculate_callbacks! (11)
function cb_button(src, ~) (12)
    disp(src.get('Text'))
end


%% itests!

%%% iexcluded_props! (13)
[ElementWithFigure.PANEL ElementWithFigure.BUTTONS]

%%% itest! (14)
%%%% iname!
Remove Figures
%%%% icode!
warning('off', [BRAPH2.STR ':ElementWithFigure'])
assert(length(findall(0, 'type', 'figure')) == 4)
```

(2) checks whether the figure still exists, otherwise (3) erases it so that (4) recreates it.

(5) ensures that `FIG` is the parent of the panel.

(6) checks whether the panel still exists, otherwise (7) erases it so that (8) recreates it.

(9) ensures that `PANEL` is the parent of each button.

(10) defines the same callback for all buttons.

(11) The callbacks are defined in the token `icalculate_callbacks!`.

(12) All callbacks have two parameters at least, corresponding to the source of the callback `src` and to its event (here, not used).

(13) The token `iexcluded_props!` determines which props to exclude from testing. Often evanescent handle and handlelist properties need to be excluded from the unit testing.

(14) This test removes the figures left over from the basic unit testing. It is good practice to ensure that no figures are left over at the end of the unit testing.

```
87 delete(findall(0, 'type', 'figure'))
88 warning('on', [BRAPH2.STR ':ElementWithFigure'])
89
90 %%% !test!
91 Simple test
92 %%%% !code!
93 ef = ElementWithFigure()
94
95 ef.memorize('BUTTONS') (15)
96
97 close(ef.get('FIG')) (16)
```

(15) memorizes the prop `BUTTON`, which in turn memorizes the props `PANEL` and `FIG`.
(16) closes the figure created in this test to ensure that no figures are left over at the end of the unit testing.