

General Developer Tutorial for BRAPH 2.0

The BRAPH 2 Developers

October 30, 2023

The software architecture of BRAPH 2.0 provides a clear structure for developers to understand and extend its functionalities. All objects (*elements*) in BRAPH 2.0 are derived from the base object `Element`. Developers can easily add new elements by writing the new elements in the simplified BRAPH 2.0 pseudocode. By recompiling BRAPH 2.0, the new elements and their functionalities are immediately integrated, also into the graphical user interface. In this Developer Tutorial, we will explain how BRAPH 2.0 is compiled, how the elements are structured, and how new elements can be implemented.

Contents

<i>Compilation and Element (Re)Generation</i>	2
<i>Elements</i>	3
<i>Setting Props</i>	7
<i>Getting Props</i>	8
<i>Memorizing Props</i>	9
<i>Element tokens</i>	10
<i>Overview of Elements</i>	12
<i>Implementation of an Element</i>	14
<i>Concrete Element extension</i>	14
<i>Results, Data, Parameters</i>	14
<i>Query</i>	15
<i>Evanescent, Gui, Figure</i>	15

Compilation and Element (Re)Generation

BRAPH 2.0 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 2.0 pseudocode, which simplifies and streamlines the coding process. To convert them into usable matlab objects, BRAPH 2.0 needs to be compiled, which is done by calling the script `braph2genesis`, which will compile the whole code, as shown in Code 1.

Code 1: Compilation of BRAPH 2.0. Executing the script `braph2genesis` compiles BRAPH 2.0 and , subsequently, unit tests it. Importantly, this function might take several hours to run (plus several more hours to unit test the compiled code).

```
1 >> 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 constants are hard-coded in the executable code to further optimize the run time.

Because this multi-stage compilation, it is not always possible to regenerate a single element without regenerating the whole BRAPH 2.0. 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 2.0 compilation and their list of props has not been altered (e.g., renamed, moved, added). In this case, it is necessary to recompile BRAPH 2.0 with `braph2genesis`.

```
1 >> close all; delete(findall(0, 'type', 'figure')); clear all ①
2
3 >> regenerate(regenerate('/src/gui', {'Pipeline'})) ②
4
5 >> regenerate(regenerate('/src/gui', {'Pipeline'}, 'DoubleCompilation',
6     false) ③
7 >> regenerate(regenerate('/src/gui', {'Pipeline'}, 'CreateElement', false)
8     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.

```

8
9 >> regenerate(regenerate('/src/gui', {'Pipeline'}, 'CreateLayout', false) (5)
10
11 >> regenerate(regenerate('/src/gui', {'Pipeline'}, 'CreateTest', false) (6)
12
13 >> regenerate(regenerate('/src/gui', {'Pipeline'}, 'UnitTest', false) (7)
14
15 >> regenerate(regenerate('/src/gui', {'Pipeline'}, 'CreateLayout', false, '
    UnitTest', false) (8)
16
17 >> regenerate(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'))`.

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

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:

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

```
el.checked(pointer)
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.

- 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.

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.

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 adjacency 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.

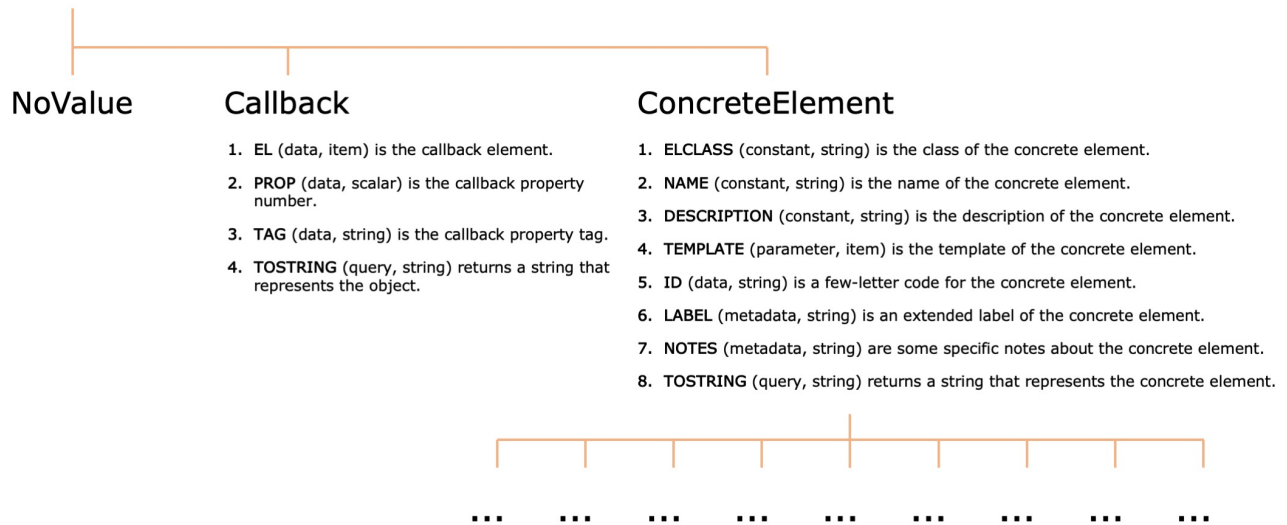
SIZE Size represents the size of a graphical component. It is a positive number (default = 1).

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`, typically one uses its subclasses and does not have any props. Its three direct subclasses are `NoValue`, `Callback`, and `ConcreteElement`, as shown in Figure 1.

Element



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`, `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()`.

```
1 Element.getNoValue()
```

No element can be a subclass of `NoValue`.

A `Callback` refers to a prop of another element `e1`, 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 `e1` with the code shown below, instead of creating new callback instances using its constructor.

```
1 e1.getCallback('PROP', PROP_NUMBER)
2 e1.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

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`.

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.

```

1 el.set('ID', 'new el id') ①
2 el.set(5, 'new el id') ②
3
4 el.set( ... ③
5   'ID', 'new el id', ...
6   'LABEL', 'new el label', ...
7   7, 'new el notes' ...
8 )
9
10 el = el.set('ID', 'new el id') ③

```

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.

③ 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 `iconditioning!`.

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 `ipreset!`.

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 `icheckProp!`.

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.

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

If an attempt is made to set a locked property, no setting occurs and a warning is thrown with warning id BRAPH2:<Element Class>.

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 `ipostset!`.

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 `ipostprocessing!`.

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 8.

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

```

1 value = el.get('ID'); ①
2
3 value = el.get(ConcreteElement.ID); ②
4
5 el.get('ID') ③
6 el.get(ConcreteElement.ID) ④
7
8 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.

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

Memorizing Props

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

Code 8: Getting a prop. This script illustrates various ways in which the value of a prop can be retrieved.

```
1 value = el.memorize('ID'); ①
2 value = el.memorize(ConcreteElement.ID); ②
3
4 el.memorize('ID') ③
5 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 typically starts with `_`. The token `iheader!` is required, while the rest is optional.

```

1 %% iheader!
2 <class_name> < <superclass_name> (<moniker>, <descriptive_name>) <header_description>.
3 %% iclass_attributes!
4 Class attributes is a single line, e.g. Abstract = true, Sealed = true.
5 %% idescription!
6 This is a plain description of the element.
7 It can occupy several lines.
8 %% iseealso!
9 Related functions and classes in a single line, coma-separated and without fullstop.
10
11 %% iconstants!
12 Constants.
13
14 %% iprops!
15 %% iprop!
16 <tag1> (<category>, <format>) <description>.
17 %%% isettings!
18 Prop settings, depending on format.
19 %%% ndefault!
20 Prop default value (seldom needed).
21 %%% iconditioning!
22 Code to condition value (before checks and calculation).
23 Can be on multiple lines.
24 The prop value is in the variable 'value',
25 where also the conditioned prop value is returned.
26 %%% ipreset!
27 Code to preset element (before checks and calculation).
28 Can be on multiple lines.
29 The prop value is in the variable 'value',
30 where also the preset prop value is returned.
31 %%% icheck_prop!
32 Code to check prop format (before calculation).
33 Can be on multiple lines.
34 The prop value is in the variable 'value'.
35 The outcome should be in variable 'check'.
36 %%% ipostset!
37 Postset code (executed after setting, but before checking, value),
38 executed on ONLY the set property.
39 Can be on multiple lines.
40 Does not return anything.
41 %%% ipostprocessing!
42 Postprocessing code (executed after setting, but before checking,
43 value), executed on ALL unlocked props after each set operation.
44 Can be on multiple lines.
45 Does not return anything.
46 %%% icheck_value!
47 Code to check prop value (after calculation).
48 Can be on multiple lines.
```

```

49     The prop value is in the variable 'value'.
50     The outcome should be in variable 'check' and the message in 'msg'.
51     %%% icalculate!
52     Code to calculate prop results (only for category RESULT).
53     Can be on multiple lines.
54     Can include callbacks as {@cb_get, 'TAG', varargin} and
55     {@cb_set, 'TAG1', value1, ...}.
56     The result should be in variable 'value'.
57     %%% icalculate_callbacks!
58     Callbacks to be used in calculate, typically as functions
59     cb_name(src, event).
60     Can be on multiple lines.
61     %%% igui!
62     GUI code for representing the panel of the prop.
63     Can be on multiple lines.
64     Should return a PanelProp object in 'pr'.
65     %%% iprop!
66     <tag2> ...
67
68 %%% iprops_update!
69 %%% iprop!
70 <tag1> (<category>, <format>) <description>. [Only description can be different from original prop]
71 %%% isettings!
72     Updated settings.
73 %%% idefault!
74     Updated default.
75 %%% iconditioning!
76     Update value conditioning (before checks and calculation).
77 %%% ipreset!
78     Update element value preset (before checks and calculation).
79 %%% icheck_prop!
80     Updated check prop format (before calculation).
81 %%% ipostset!
82     Update postset (after setting, but before checking, value).
83 %%% ipostprocessing!
84     Update value postprocessing (after setting, but before checking, value).
85 %%% icheck_value!
86     Updated check prop value (after calculation).
87 %%% icalculate!
88     Updated calculation.
89 %%% icalculate_callbacks!
90     Updated calculate callbacks.
91 %%% igui!
92     Updated GUI.
93 %%% iprop!
94     <tag2> ...
95
96 %%% igui!
97 %%% imenu_import!
98     Menu Import for the GUI figure.
99     The element is el.
100    The menu is menu_import.
101    The plot element is pe.
102 %%% imenu_export!
103    Menu Export for the GUI figure.
104    The element is el.
105    The menu is menu_export.
106    The plot element is pe.
107
108 %%% ilayout!
109 %%% iprop!

```

```

110   %% iid!
111   Prop id, e.g., Element.TAG, ordered as they should appear.
112   %% ititle!
113   String containing the title of the prop panel.
114   %% iprop!
115   ...
116
117   %% itests!
118   %% iexcluded_props!
119   Row vector with list of props to be excluded from standard tests.
120   %% iwarning_off!
121   Switches off the warnings regarding the element.
122   %% itest!
123   %% iname!
124   Name of the text on a single line.
125   %% iprobability!
126   Probability with which this test is performed. By default it is 1.
127   %% icode!
128   Code of the test.
129   Can be on multiple lines.
130   %% itest!
131   ...
132   %% itest_functions!
133   Functions used in the test.
134   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 2.0 (gray shaded area) includes the compiler (genesis), the essential source code (src), and the essential functionalities for

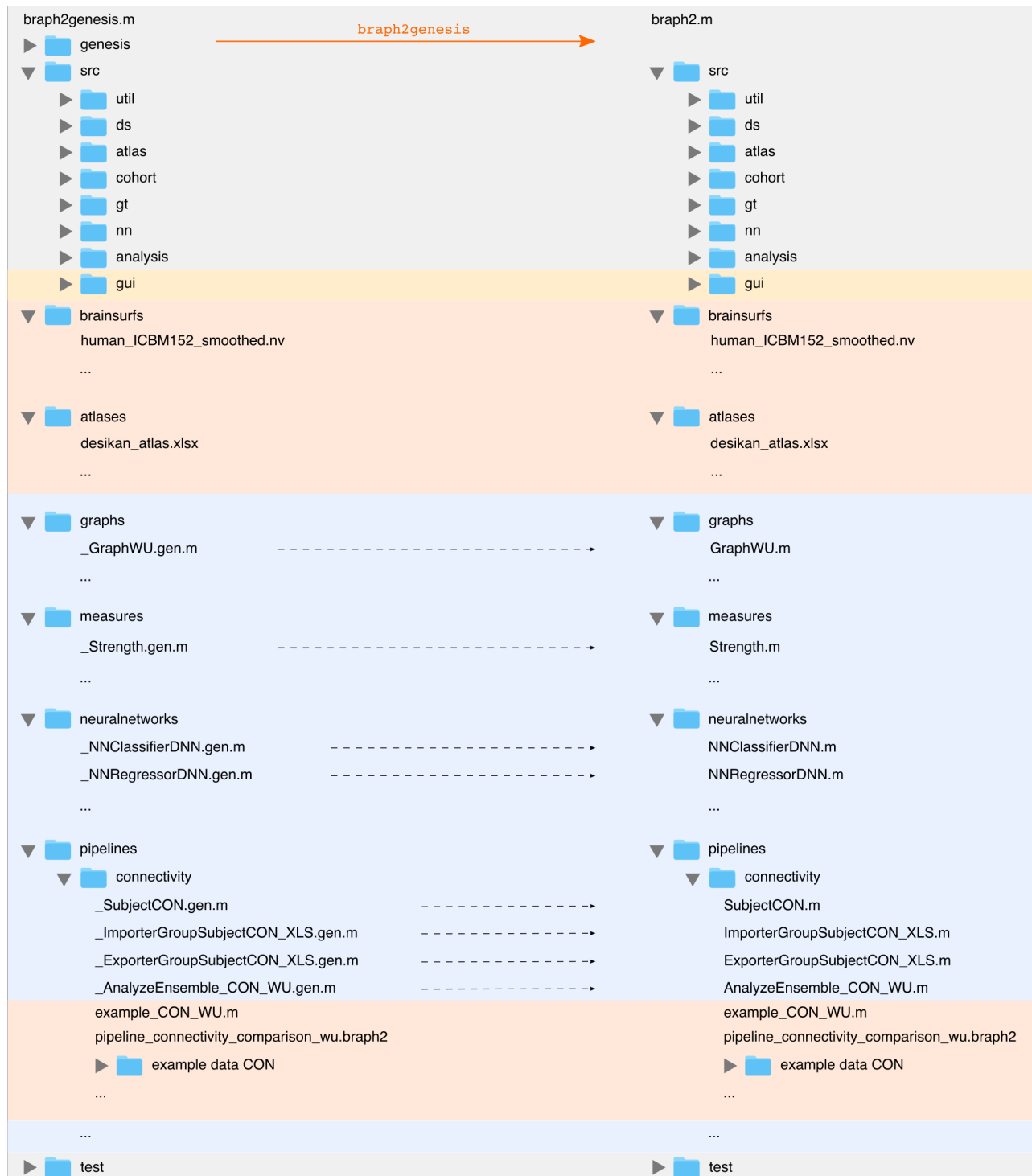


Figure 2: **BRAPH 2.0 genesis..** Directory structure of `braph2genesis` (left) and `braph2` (right).

the GUI (gui, yellow-shaded area). The users can easily add new 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 2.0 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 2.0

To speed up the compilation of BRAPH 2.0 when trying these examples, it is possible to perform a light version of the compilation using the script `braph2genesis_with_exclusions`, which permits one to exclude entire folders or individual elements, as shown in Code 11.

Code 11: BRAPH 2.0 genesis with exclusions. Using `braph2genesis_with_exclusions`, it is possible to exclude some folders and elements, which are defined in the variable `excluded`. You can place your elements in the folder `util`.

```

1 ...
2 excluded = { ...
3   'gt', 'atlas', 'cohort', 'analysis', 'nn', 'gui', ...
4   'brainsurfs', 'atlases', 'graphs', 'measures', 'neuralnetworks', 'pipelines', ...
5   '_Exporter.gen.m', '_Importer.gen.m' ...
6 };
7 ...

```

This same approach (appropriately altering the 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 2.0 is saved in `brap2_with_exclusions`, which is ignored by GIT.

A Simple Calculator

We will now create our first element (Code ??), 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 directly deriving from ConcreteElement.

```

1 %% iheader! ①
2 ArithmeticOperations < ConcreteElement (ao, arithmetic operation calculator)
   calculates simple arithmetic operations.
3
4 %%% idescription!
5 An Arithmetic Operation Calculator (ArithmeticOperations) contains two
6 numbers as data scalar props and calculates their sum and difference as
7 result scalar props.
8
9 %%% iseealso!
10 LogicalOperations, GeometricalOperations
11
12
13 %% iprops_update! ②
14
15 %%% iprop!
16 ELCLASS (constant, string) is the class of the arithmetic operation
   calculator.
17 %%% idefault!
18 'ArithmeticOperations' ③
19
20 %%% iprop!
21 NAME (constant, string) is the name of the arithmetic operation calculator.
22 %%% idefault!
23 'Arithmetic Operation Calculator'
24
25 %%% iprop!
26 DESCRIPTION (constant, string) is the description of the arithmetic
   operation calculator.
27 %%% idefault!
28 'An Arithmetic Operations element (ArithmeticOperations) contains two
   numbers as data scalar props and calculates their sum and difference as
   result scalar props.'
29
30 %%% iprop!
31 TEMPLATE (parameter, item) is the template of the arithmetic operation
   calculator.
32 %%% isettings!
33 'ArithmeticOperations' ④
34
35 %%% iprop!
36 ID (data, string) is a few-letter code for the arithmetic operation
   calculator.
37 %%% idefault!
38 'ArithmeticOperations ID'
39
40 %%% iprop!
41 LABEL (metadata, string) is an extended label of the arithmetic operation
   calculator.
42 %%% idefault!
43 'ArithmeticOperations label'
44
45 %%% iprop!
46 NOTES (metadata, string) are some specific notes about the arithmetic
   operation calculator.
47 %%% idefault!
48 'ArithmeticOperations notes'

```

① The iheader! token is the only required one.

② The iprops_update! token permits to update the properties of the ConcreteElement. The updated parts have been highlighted.

③ must be the name of the element.

④ must be the name of the element.

```

49
50 %% iprop! ⑤
51 TOSTRING (query, string) returns a string that represents the arithmetic
    operation calculator.
52 %%% icalculate! ⑥
53 a = ao.get('A');
54 b = ao.get('B');
55 value = ['Calculator of the sum and difference of ' num2str(A) ' and '
    num2str(B)];
56
57
58 %% iprops! ⑦
59
60 %% iprop! ⑧
61 A (data, scalar) is the first number.
62
63 %% iprop! ⑨
64 B (data, scalar) is the second number.
65
66 %% iprop! ⑩
67 SUM (result, scalar) is the sum of the two numbers (A + B).
68 %%% icalculate! ⑪
69 value = ao.get('A') + ao.get('B');
70
71 %% iprop! ⑫
72 DIFF (result, scalar) is the difference of the two numbers (A - B).
73 %%% icalculate! ⑬
74 value = ao.get('A') - ao.get('B');
75
76
77 %% itests! ⑭
78
79 %% itest!
80 %%% iname!
81 Simple test
82 %%% icode!
83 ao = ArithmeticOperations('A', 6, 'B', 4)
84
85 string = ao.get('TOSTRING')
86 assert(~ao.isLocked('A')) ⑮
87 assert(~ao.isLocked('B')) ⑯
88
89 sum = ao.get('SUM')
90
91 assert(ao.isLocked('A')) ⑰
92 assert(ao.isLocked('B')) ⑱
93
94 diff = ao.get('DIFF')
95
96 sum_raw = ao.getr('SUM') ⑲
97 diff_raw = ao.getr('DIFF') ⑳
98 assert(isa(sum_raw, 'NoValue') && isa(diff_raw, 'NoValue'))
99

```

⑤ Often, it is not necessary to updated TOSTRING, as the default works for most cases.

⑥ returns the string, which must be saved in the variable value.

⑦ The iprops! token permits to add additional props.

⑧ and ⑨ are two data props.

⑩ is a result prop.

⑪ calculates the sum of the two numbers. The result must be saved in the variable value.

⑫ is a result prop.

⑬ calculates the difference of the two numbers. The result must be saved in the variable value.

⑭ The itests! token permits to add unit tests.

⑮ and ⑯ Both props A and B are not locked, even though the query prop TOSTRING has been calculated.

⑰ and ⑱ Both props A and B are now locked, because the result prop SUM has been calculated. From now on their value cannot be changed.

⑲ and ⑳ Note that both the result props SUM and DIFF are NoValue, because they have not been memorized yet.


```

100 %%% itest! ②1
101 %%% iname!
102 Simple test with memorization
103 %%% icode!
104 ao = ArithmeticOperations('A', 6, 'B', 4)
105
106 sum = ao.memorize('SUM')
107 diff = ao.memorize('DIFF')
108
109 sum_raw = ao.getr('SUM')
110 diff_raw = ao.getr('DIFF')
111 assert(~isa(sum_raw, 'NoValue') && ~isa(diff_raw, 'NoValue'))

```

②1 alters the previous test to memorize the results.

Calculator with Seeded Randomness

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

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

```

1 %% iheader!
2 SeededRandomness < ConcreteElement (sr, randomizer) generates a random
   number.
3
4 %%% idescription!
5 ... ①
6
7
8 %% iprops_update!
9
10 %%% iprop!
11 ELCLASS (constant, string) is the class of the randomizer.
12 %%% idefault!
13 'SeededRandomness'
14
15 ... ②
16
17
18 %% iprops!
19
20 %%% iprop!
21 RANDOM_NUMBER (result, scalar) is a random number.
22 %%% icalculate!
23 value = rand();
24
25
26 %% itests!
27
28 %%% itest!
29 Simple test
30 %%% icode!
31 sr1 = SeededRandomness()
32 sr2 = SeededRandomness()
33
34 assert(sr1.get('RANDOM_NUMBER') == sr1.get('RANDOM_NUMBER')) ③
35 assert(sr2.get('RANDOM_NUMBER') == sr2.get('RANDOM_NUMBER')) ⑤
36 assert(sr1.get('RANDOM_NUMBER') ~= sr2.get('RANDOM_NUMBER')) ⑥

```

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).

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 different randomizers return different values.

Query

We can now demonstrate the use of query props by expanding the `ArithmeticOperations` (Code ??).

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

```

1 %% iheader!
2 ArithmeticOperationsWithQuery < ArithmeticOperations (ao, calculator with
   query) calculates simple arithmetic operations with a query.
3
4 %%% idescription!
5 . . .
6
7
8 %% iprops_update!
9
10 %%% iprop!
11 ELCLASS (constant, string) is the class of the calculator with query.
12 %%% idefault!
13 'ArithmeticOperationsWithQuery'
14
15 . . .
16
17
18 %% iprops!
19
20 %%% iprop!
21 SUM_OR_DIFF (query, scalar) returns the sum or difference depending on the
   argument.
22 %%% icalculate!
23 % R = ao.get('SUM_OR_DIFF', SUM_OR_DIFF) returns the sum of A and B if ①
24 % SUM_OR_DIFF = 'SUM' or the difference of A and B if SUM_OR_DIFF = 'DIFF'.
25
26 if isempty(varargin) ②
27     value = NaN;
28     return
29 end
30 sum_or_diff = varargin{1};
31
32 switch sum_or_diff
33     case 'SUM'
34         value = ao.get('SUM');
35
36     case 'DIFF'
37         value = ao.get('DIFF');
38
39     otherwise
40         value = NaN;
41 end
42
43
44
45 %%% itests!

```

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.

```

46
47 %% itest!
48 Simple test
49 %%% icode!
50 ao = ArithmeticOperationsWithQuery('A', 6, 'B', 4)
51

```

```

52 assert(ao.get('SUM_OR_DIFF', 'SUM') == ao.get('SUM')) (3)
53 assert(ao.get('SUM_OR_DIFF', 'DIFF') == ao.get('DIFF')) (4)
54 assert(isnan(ao.get('SUM_OR_DIFF'))) (5)
55 assert(isnan(ao.get('SUM_OR_DIFF', 'anything else'))) (6)

```

(3) and (4) returns the sum or the difference depending on the argument.

(5) and (6) returns NaN when the input is absent or unexpected.

Evanescent, Gui, Figure

We can now demonstrate the use of evanescent props and graphical handles (Code ??).

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
8 %% iprops_update!
9
10 %%% iprop!
11 ELCLASS (constant, string) is the class of the element with figure.
12 %%% icalculate!
13 'ElementWithFigure'
14
15 ...
16
17
18 %% iprops!
19
20 %%% iprop!
21 FIG (evanescent, handle) is the handle of a figure.
22 %%% icalculate!
23 value = uifigure( ... (1)
24     'Name', 'Figure from ElementWithFigure', ...
25     'Color', BRAPH2.COL ...
26 );
27
28 %%% iprop!
29 PANEL (evanescent, handle) is the handle of the panel.
30 %%% icalculate!
31 if ~check_graphics(ef.memorize('FIG'), 'figure') (2)
32     ef.set('FIG', Element.getNoValue()); (3)
33 end
34
35 fig = ef.memorize('FIG'); (4)
36
37 value = uipanel( ...

```

(1) renders a figure and returns its handle.

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

(4) recreates it.

```

38     'Parent', fig, ... ⑤
39     'Units', 'normalized', ...
40     'Position', [.25 .25 .50 .50], ...
41     'BackgroundColor', BRAPH2.COL_BKG ...
42 );
43
44 %%% iprop!
45 BUTTONS (evanescent, handlelist) is the list of handles of the buttons.
46 %%% icalculate!
47 if ~check_graphics(ef.gettr('PANEL'), 'uipanel') ⑥
48     ef.set('PANEL', Element.getNoValue()); ⑦
49 end
50
51 panel = ef.memorize('PANEL'); ⑧
52
53 value = {};
54 for i = 1:1:10
55     value{i} = uibutton( ...
56         'Parent', panel, ... ⑨
57         'Text', ['B' int2str(i)], ...
58         'Position', [ ...
59             (i - 1) * w(panel, 'pixels') / 10 ...
60             (i - 1) * h(panel, 'pixels') / 10 ...
61             w(panel, 'pixels') / 10 ...
62             h(panel, 'pixels') / 10 ...
63         ], ...
64         'ButtonPushedFcn', {@cb_button} ... ⑩
65     );
66 end
67 %%% icalculate_callbacks! ⑪
68 function cb_button(src, ~) ⑫
69     disp(src.get('Text'))
70 end
71
72
73 %%% itests!
74
75 %%% iexcluded_props! ⑬
76 [ElementWithFigure.PANEL ElementWithFigure.BUTTONS]
77
78 %%% itest! ⑭
79 %%% iname!
80 Remove Figures
81 %%% icode!
82 warning('off', [BRAPH2.STR ':ElementWithFigure'])
83 assert(length(findall(0, 'type', 'figure')) == 4)
84 delete(findall(0, 'type', 'figure'))
85 warning('on', [BRAPH2.STR ':ElementWithFigure'])
86
87 %%% itest!
88 Simple test
89 %%% icode!
90 ef = ElementWithFigure()
91
92 ef.memorize('BUTTONS') ⑮
93

```

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.

⑮ memorizes the prop BUTTON, which in turn memorizes the props PANEL and FIG.

94 `close(ef.get('FIG'))` 16

16 closes the figure created in this test to ensure that no figures are left over at the end of the unit testing.