StrEmbed-6-1 Users' Manual

Embedding design structures in engineering information

Hugh Rice ([h.p.rice@leeds.ac.uk](mailto:h.p.rice@leeds.ac.uk)) and Hau Hing Chau ([h.h.chau@leeds.ac.uk](mailto:h.h.chau@leeds.ac.uk))

University of Leeds

2021-2023

What is StrEmbed-6-1?

Structure Embedding version 6-1 (StrEmbed-6-1) is a deliverable from the Design Configuration Spaces (DCS) project, funded by the Engineering and Physical Sciences Research Council (grant reference EP/N005694/1).

Engineers use design structures, such as bills of materials (BoMs), to tailor product definitions, including shape, for particular activities. For example, an engineering BoM defines the as-designed product whereas a manufacturing BoM defines the as-built state of the same product and a service BoM includes information on how the product has been maintained. All of these BoMs relate to the same designed product. However, in practice, because of restrictions arising from current computer-aided design technologies and associated business systems, different BoMs are usually related to separate digital definitions of the same product. This creates significant data management problems that add cost, time and rework into product development processes. If resolved, substantial business benefits, through improved efficiency and effectiveness of product development processes, could be achieved.

StrEmbed-6-1 allows assembly/BoM structures to be (a) loaded from STEP/STP files or custom StrEmbed data files and (b) manipulated *via* a number of operations: aggregate, disaggregate, remove part, add part, flatten, collapse and shuffle/sort/move parts or sub-assemblies. The assembly structure is visualised as a tree/directory, a lattice/network and as images (for part selection) and a 3D CAD-style viewer. This functionality allows the user to create unlimited new BoMs from (a) existing ones or (b) from scratch.

The software represents a step towards a full assembly creation and comparison tool that will allow a design structure to be embedded onto a corresponding order lattice. An order lattice is representation of a partially ordered set (poset) of the constituent parts of a product – the partial ordering being part-whole relationships between parts and sub-assemblies – which is equivalent to a Boolean algebra. This algebra has a number of useful mathematical properties that allow generation of alternative design structures for each stage of product life cycle, *e.g.* “as designed”, “as manufactured” and “as maintained”. This embedding relationship can be visualised using a Hasse diagram, and any combination or permutation of an alternative design structure can embedded in the same lattice.

In addition, StrEmbed allows: (a) design structures to be exported to a common data format, for use in other CAD systems; and (b) reconciliation of multiple BoMs of the same product that have diverged during the product development process, *i.e.* using StrEmbed, the user can automatically link common items (parts and sub-assemblies) across multiple BoMs, which can then be embedded in a common data structure. Both individual assemblies and whole projects, which may contain multiple BoMs, can be exported to StrEmbed-specific assembly (.asy) files (single assemblies only) or Microsoft Excel files for further analysis.

How to install and run StrEmbed

Software details and operating system dependence

StrEmbed is written in Python 3.9 and, as such, can be run on any operating system. However, it has only been tested on Microsoft Windows 10 64-bit, and the executable provided (see below) is Windows-specific. Please contact the developers if you encounter problems when running StrEmbed-6-1 on an operating system other than Windows 10 64-bit.

Additionally, if you use another operating system or a version of Python 2.x and have difficulty running StrEmbed, please contact the authors. StrEmbed relies on a large number of packages (dependencies), the principal of which being: the cross-platform GUI interface module [Wxpython](https://www.wxpython.org/) that relies on or mimics the native controls of the user’s operating system; and a 3D viewer based on the [Open Cascade](https://www.opencascade.com/) framework *via* the module [PythonOCC](https://github.com/tpaviot/pythonocc).

Running StrEmbed as an executable (recommended) and creating your own

Download the “StrEmbed.exe” file from the [Github repository](https://github.com/paddy-r/StrEmbed-6-1), *via* the “Releases” page (*via* link on right of repository home page). This is an executable file that can be run directly and does not require any additional installations. The executable was created using the [Pyinstaller](https://www.pyinstaller.org/) module, and is OS-specific (currently Windows only). You can create an executable for your particular OS by running Pyinstaller yourself, using the source files provided at the Github repository, *via* the process below.

Running StrEmbed as script

Firstly, create and activate a suitable Python environment. We recommend Anaconda/Miniconda, and the file environment.yml is provided for this purpose. Execute the following commands, choosing a name for your environment.

conda env create --name <env\_name> -f environment.yml

conda activate <env\_name>

Next, download the StrEmbed repository from Github [here](https://github.com/paddy-r/StrEmbed-6-1), or clone it using the command git clone <https://github.com/paddy-r/StrEmbed-6-1>, ensuring you have Git installed. It is also necessary to clone and install the PartFind package, a dependency of StrEmbed, as follows:

pip install git+https://github.com/thazlehurst/partfind@strembed\_legacy

Then, navigate to the uppermost directory of StrEmbed and either run StrEmbed directly in the command line (Windows), as follows:

python StrEmbed.py

or in your Python IDE.

Build your own StrEmbed executable (not recommended)

The following instructions are for creating your own executables using Pyinstaller. First, install Pyinstaller itself in your virtual environment, as follows:

pip install pyinstaller

At time of writing, it is then necessary to update Pyinstaller using a development version, to solve issues related to the [Scikit-learn](https://scikit-learn.org/stable/) package, as described in issues at Github [here](https://github.com/pyinstaller/pyinstaller/issues/6916) and [here](https://github.com/pyinstaller/pyinstaller/issues/6919), using the following command, which should install the "pyinstaller-hooks-contrib" package at version 2022.8 or higher.

pip install git+https://github.com/pyinstaller/pyinstaller-hooks-contrib.git

The Pyinstaller module allows detailed specifications of the build process and you are referred to its documentation, but some examples of commands that can be used to generate an executable are provided in create\_exe.txt. It is recommended to move your executable to a folder with a short path name, *e.g.* the desktop, as longer paths sometimes cause file-not-found errors or similar. This is not an issue with StrEmbed or Python.

You may add the following optional commands to the Pyinstaller build command: -F (or --onefile), which means a single executable is created, without any additional folders of libraries, *etc.;* and -w (or --noconsole) creates an executable without an accompanying console window for debugging. If you do not use the -F option, your executable will be in the "dist" directory. You can then double-click the StrEmbed.exe executable to run it. If StrEmbed fails to run, you can print the output from your executable to the console and save it to a text file as below, where <file\_name> is whatever you choose; please submit an issue via Github in this case.

StrEmbed.exe > <file\_name>.txt & <file\_name>.txt

Using StrEmbed

Export a STEP AP214 file from another CAD system

* Create or open an assembly in another CAD system, for example [SolidWorks](https://www.solidworks.com/) or the free, open-source application [FreeCAD](https://www.freecadweb.org/).
* Export the assembly to a STEP AP214 file, ensuring the file extension is .STP or .STEP or the lower-case equivalent.

Read a STEP file or .asy file using StrEmbed

* Run StrEmbed by any of the methods described in the previous section.
* Open a file via the “File open” tool and then select a STEP or STP file to open, as shown in Figure 1. The example used hereafter for illustration is “Torch Assembly.STEP”, which consists of a simple torch of ten distinct parts and is bundled with StrEmbed. Some other examples are also presented. Alternatively, you may right-click on the tab of the active assembly and select “Import StrEmbed assembly from file” to import an assembly from a previous session.

|  |  |
| --- | --- |
|  | C:\Users\prehr\OneDrive - University of Leeds\__WORK SYNCED\_Work\_DCS project\__ALL CODE\_Repos\StrEmbed-6-1\_6-1 users manual and notes\fileopen.png |
| Figure 1: Opening a STEP or STP file. Left: *via* tool in “Home” tab. Right: file dialog. | |

* After a STEP file is loaded, the corresponding assembly tree structure is displayed in the “parts view” on the left and as a Hasse diagram (a kind of directed graph visualisation) in the “lattice view” on the right. Images of parts and assemblies are displayed as toggle buttons in the “selector view” in the top-centre when the corresponding check boxes in the parts view are activated. The “3D view” in the bottom-centre shows (a) all parts selected in any of the other views, or (b) all parts in the assembly, if none is selected elsewhere. The user interface with the torch example loaded is shown in Figure 2.

|  |
| --- |
|  |
| Figure 2: StrEmbed user interface with torch example shown. |

* Additional functionality is provided in the lattice view, allowing the user to control which part of the network is visible, and to save images, for example, *via* [Matplotlib](https://matplotlib.org/)’s [FigureCanvas](https://matplotlib.org/tutorials/intermediate/artists.html). In the 3D view, the user can modify the view of parts with the mouse as follows:
  + Left button: hold to rotate
  + Middle button: hold to move
  + Right button: hold to zoom
* Parts and assemblies can be selected in the parts, selector and lattice views: in the parts view by single left mouse click (and multiple items can be selected by holding the SHIFT or CTRL keys); in the selector view by single left click (which toggles the selected item); and in the lattice view by right or left single click. Selections in one view are reflected immediately in the others.
* Please note that redundant subassemblies (*i.e.* any with only one child) are removed from the assembly structure upon loading from a STEP file, or upon modification of the assembly structure (see next section) by the user *via* StrEmbed’s interface, *e.g.* deleting a node and thereby leaving a subassembly with a single child. This is to ensure operations on the assembly adhere to rules dictated by order lattice theory.

Modifying assembly structure, *i.e.* BoM transformation

* A variety of operations can be performed on any parts and assemblies that are selected, either via the toolbar icons or via pop-up menu activated by right-clicking on selected items in the parts view. These operations are described in Table 1.

| Table 1*:* Toolbar operations. | | |
| --- | --- | --- |
| *Operation* | *Description* | *Toolbar icon* |
| *“Home” tab* | | |
| Assemble | Construct a new sub-assembly of all selected items | C:\Users\prehr\.spyder-py3\_GUI development\Images\assemble.bmp |
| Flatten | Remove all sub-assemblies from within the selected item, creating a flat list | C:\Users\prehr\.spyder-py3\_GUI development\Images\flatten.bmp |
| Disaggregate | Create assembly from single selected part | C:\Users\prehr\.spyder-py3\_GUI development\Images\disaggregate.bmp |
| Aggregate | Reduce sub-assembly to single item by removing all children; child IDs are retained for later use | C:\Users\prehr\.spyder-py3\_GUI development\Images\aggregate.bmp |
| Add node | Add new item to a selected sub-assembly | C:\Users\prehr\.spyder-py3\_GUI development\Images\add_node.bmp |
| Remove node(s) | Remove selected item(s) | C:\Users\prehr\.spyder-py3\_GUI development\Images\remove_node.bmp |
| Toggle sort type | Toggle between sorting child items by unique ID (default) or alphabetically according to description | C:\Users\prehr\.spyder-py3\_GUI development\Images\sort_mode.bmp |
| Reverse sort | Toggle between forward (default) and backward sort order; does not change sort type | C:\Users\prehr\.spyder-py3\_GUI development\Images\sort_reverse.bmp |
| *“Assistant” tab* | | |
| Map assembly elements | Create associations between parts in two assemblies based on pair-wise similarity, and seek user input where associations are not clear. (Not yet implemented.) | C:\_Work\_DCS project\__ALL CODE\_Repos\StrEmbed-5-5\StrEmbed-5-5 for git\Images\injection.png |
| Calculate similarity | Calculate the similarity, which is a user-defined aggregate of part data, assembly structure and part geometry, between two assemblies. (Not yet implemented.) | C:\_Work\_DCS project\__ALL CODE\_Repos\StrEmbed-5-5\StrEmbed-5-5 for git\Images\compare.png |
| Reconcile assemblies | Find common items between two assemblies *via* BoM reconciliation (see next section) | C:\Users\prehr\.spyder-py3\_GUI development\StrEmbed-5-3\Images\tree_small.bmp |
| Suggest new assembly | Create new assembly based on user preferences and machine learning technology. (Not yet implemented.) | C:\_Work\_DCS project\__ALL CODE\_Repos\StrEmbed-5-5\StrEmbed-5-5 for git\Images\bulb_sharp.png |

* Figure 3 and Figure 4 show examples of assembly operations performed via pop-up menus, before and after the operations. The changes are reflected immediately in the lattice view.

|  |  |
| --- | --- |
|  |  |
| Figure 3: “Assemble” operation via right-click pop-up menu. Left: before. Right: after. | |

|  |  |
| --- | --- |
|  |  |
| Figure 4: “Aggregate” operation via right-click pop-up menu. Left: before. Right: after. | |

* In addition to the toolbar/pop-up menu functionality described above, items in the parts view can be moved arbitrarily by dragging and dropping using the left mouse button, either within the same sub-assembly (which causes the item(s) and its/their sibling(s) to be reordered as the user wishes) or to another sub-assembly (which re-parents the dragged item(s)).
* Lastly, item descriptions can be modified by slow double left-clicking on the item text in the parts view.
* Some tools in the Assistant tab are not yet implemented.

Linking common items between assemblies, *i.e.* BoM reconciliation

BoM reconciliation consists of automatically identifying common items (parts and sub-assemblies) between multiple assemblies and creating persistent links between them, and StrEmbed provides the functionality to do so. In practice, StrEmbed performs BoM reconciliation by comparing each item in two BoMs – for example, if BoM 1 has 10 parts and BoM 2 has 20, it will be necessary to make 200 comparisons – and finding the optimal set of matches. The characteristics of the two BoMs that are used in each comparison are: (1) part names, (2) the assembly structure local to the part (its parent’s name, its number of siblings and its depth within the assembly structure), (3) the volume of the part (as a proxy for its overall size) and (4) the part’s topology, as measured by the machine learning-trained tool, PartFind, which is also part of the DCS project and can be found at [Github here](https://github.com/thazlehurst/partfind).

Upon loading or importing an assembly, StrEmbed will ask whether you want to add the assembly to the lattice, *i.e.* the data structure that contains links between common items in the assemblies in your project. You may either click “OK”, or “Cancel”. If you click “OK” and no assemblies are present in the lattice, the assembly will be added without any further user input as no reconciliation is necessary. However, if one or more assemblies are present in the lattice, you will be presented with a BoM reconciliation dialog, as shown in Figure 5. You may then choose (a) which existing assembly you wish to compare the active assembly to, (b) whether to employ “blocking”, *i.e.* to pre-group parts with the same names or volume and (c) which of the four comparison metrics described above to use in the reconciliation process. When the comparison process has finished, you will be able to view the results in a dialog window and decide whether to add the active assembly to the lattice, shown in Figure 5. One or more of StrEmbed’s view may then change as the lattice is updated.

Alternatively, any assembly not embedded in the lattice may be added at any time by right-clicking the active assembly’s tab and selecting “Add assembly to lattice”. An assembly may also be removed from the lattice (but not from your project overall) by right-clicking on its tab and selection “Remove assembly from lattice”; it may be added again at any time.

|  |  |
| --- | --- |
|  |  |
| Figure 5: Assembly reconciliation example. Left: BoM reconciliation specification dialog. Right: BoM reconciliation report dialog (example shown: two instances of torch example). | |

Saving individual assemblies and exporting whole projects

StrEmbed allows your project to be saved or exported in two ways. First, you may export your whole project to a Microsoft Excel file, in which the first sheet is a summary of the project, and there is a subsequent sheet for each assembly in your project that contains part and sub-assembly information in the form of an indented list. To do so, click the second button in the “File” toolbar (see Figure 1) and specify your path and filename.

Second, you may export individual assemblies as StrEmbed-specific .asy files, which can then be loaded later. To do so, click the third button in the “File” toolbar and specify your path and filename.

People

The DCS project is hosted by the University of Leeds. Its members are Prof. Alison McKay, Prof. David Hogg, Prof. Alan de Pennington, Dr. Mark Robinson, Dr. Tiziana Callari, Dr. Hau Hing Chau, Dr. Tom Hazlehurst and Dr. Hugh Rice.

How to track and report bugs or get help

Known bugs and issues are tracked at the Github repository [here](https://github.com/paddy-r/StrEmbed-6-1). Please log bug/issue reports, request help or give feedback there or *via* the e-mail address below.

Hugh Rice

School of Geography and School of Mechanical Engineering

University of Leeds

Leeds LS2 9JT

[h.p.rice@leeds.ac.uk](mailto:h.p.rice@leeds.ac.uk)

Appendix 1: Event and view management in StrEmbed

StrEmbed uses the Wxpython module for its graphical user interface (GUI), a kind of event-driven system in which the application reacts to user’s interactions via the mouse and keyboard. StrEmbed’s GUI has multiple views, currently the parts, selector, 3D and lattice views. A user interacts with an object in one view, and the effects of the interaction must be propagated to all. This appendix describes the logic behind the process of GUI interactions in StrEmbed, specifically item selection and deselection. A number of factors must be accounted for: (a) the nature of an interaction can be modified by the user (*e.g.* holding the CTRL key in some views causes existing node selections to be retained, holding SHIFT in the parts view may cause multiple selections); (b) the effects of an interaction depends on the view (*e.g.* selections are retained by default in the lattice view but not in the others); and (c) in some views the interaction itself produces a GUI event, which must be handled correctly and not executed multiple times.

Table 2 shows the set-theoretical definitions of each kind of node in the GUI interaction process. Before a user interaction, the GUI is in its “old state” (OS). The goal is to produce a global “new state” (NS), common to all views, in which the effects of the user’s interaction have been propagated. If the user interaction is in a view that itself generates (emits) a GUI event, then, depending on the kind of interaction, there may then be an “intermediate state” (IS) that differs from the old state. In this intermediate state, some of the (de)selection operations have already been performed and so must be excluded from operations on that view later in the GUI update process.

| Table 2: Definitions of item sets for GUI selection logic. O = old, I = intermediate, N = new, S = selected, U = unselected. | | |
| --- | --- | --- |
| **General definitions** | | |
|  | *Selected nodes* | *Unselected nodes* |
| *Old state* | OS | OU |
| *Intermediate state* | IS | IU |
| *New state* | NS | NU |
| **Transitional** | | |
| *To be (un)selected (between old and new states)* | TS = NS OU | TU = NU OS |
| *Changing* | CH = TS TU | |
| *Already (un)selected (in intermediate state)* | AS = TS IS | AU = TU IU |
| *Still to be (un)selected (in intermediate state)* | SS = TS IU | SU = TU IS |
| *To be excluded from further operations in GUI element where event originated* | EX = AS AU | |
| *(Un)selected in both OS and NS, but not in IS, so requires opposite of (un)selection again* | ADS = (OS NS) IU | ADU = (OU NU) IS |
| *Additional operations in GUI element where event originated* | AD = ADS ADU | |

The identity and/or method of computation for OS, IS, NS, EX and AD are given in Table 3 for each view in StrEmbed, and it is important to note that the parts view is peculiar in that it is the “master view”, *i.e.* the view from which the selection state of the system (*i.e.* the list of selected items in the active assembly) is retrieved in general. Also, it can be seen that IS => OS for both the lattice and 3D views, as no GUI events are generated in those views, and they must be updated explicitly. Lastly, IS in the selector view requires some additional logic because a user interaction with a toggle button there generates a GUI event; however, only one item can be toggled at a time, so the additional logic is simple.

| Table 3: Identities/method of computation of key sets for each view; IDS is set of nodes that user interacted with; *selected\_items* is a method within StrEmbed that retieves all selected items from the parts view. Function *get\_NS* given in Figure 7. | | | | | |
| --- | --- | --- | --- | --- | --- |
| *View*  | *Set*  | | | | |
| OS | IS | NS | EX | AD |
| Lattice | *selected\_items* | OS | *get\_NS*(OS, IDS, *retain* = T, *select\_children* = T/F) | Empty | Empty |
| 3D | *selected\_items* | OS | *get\_NS*(OS, IDS, *retain* = T/F, *select\_children* = T/F) | Empty | Empty |
| Selector | *selected\_items* | OS – ID or OS ID (1) | *get\_NS*(OS, IDS, *retain* = T, *select\_children* = T/F) | ID | Empty |
| Parts | *selected\_items* (2) | *selected\_items* (2) | IS or IS *children*(each in IS) (3) | AS AU | ADS ADU |
| (1) Depends on whether ID (node) is already selected, because selector view toggles state.  (2) OS and IS in parts view retrieved before and after change-in-selection event and can therefore differ.  (3) NS in parts view depends on whether *select\_children* is T or F. | | | | | |

Pseudocode for the function *get\_NS*, which computes the set of nodes, NS, to be selected in the “new state” (see Table 3) is given in Figure 7.

|  |
| --- |
| function *get\_NS*(OS, IDS, *retain* = T/F, *select\_children* = T/F):  *# If interaction arises in toggle-only view*  if length(IDS) = 1:  if *retain*:  NS => copy(OS)  if (ID in OS):  NS => OS – ID  else:  NS => OS ID  if *select\_children*:  NS => NS *children*(ID)  else:  if (ID in OS):  NS => empty  else:  NS => ID  if *select\_children*:  NS => NS *children*(ID)  *# If interaction arises in view with several selection possibilities*  else if length(IDS) > 1:  if retain:  NS => copy(OS)  else:  NS => empty  for ID in IDS:  NS => NS ID  if *select\_children*:  NS => NS *children*(ID)  return NS |
| Figure 7: Pseudocode for computing set of nodes in “new state”, NS. ID is node of user interaction; OS is set of nodes in “old state”; *retain* and *select\_children* are options specifying whether existing selections to be retained, and whether children of node interacted with are also to be selected. |

A flowchart of GUI processes is shown in Figure 8, in which it is noted that there is no need to veto any lattice or 3D view events, as interactions in those views do not generate GUI events, and are updated explicitly. Pseudocode for the *update\_GUI* function (see Figure 8) is given in Figure 9, where view-specific veto flags ensure that event-emitting interactions do not cause operations in the GUI update process to be duplicated.

|  |
| --- |
|  |
| Figure 8: GUI update process flowchart, showing logic for lattice, 3D, selector and parts views (L-R). Diamonds: user interactions; squares: processes/functions; ovals: events generated by GUI event loop. |

|  |
| --- |
| function *update\_GUI*(OS, NS, \*\**kwargs*):  Compute OU, NU, CH, etc.  Update 3D view  Update lattice view  *veto\_selector* => True  Update selector view, accounting for excluded (EX) and additional (AD) items  *veto\_selector* => False  *veto\_parts* => True  Update parts view, accounting for excluded (EX) and additional (AD) items  *veto\_parts* => False |
| Figure 9: Pseudocode for globally updating GUI. \*\**kwargs* represents keyword arguments to allow events from specific views to be vetoed/excluded. |

The structure of GUI interactions and operations described in this section was designed so that additional views can easily be added, and although a general view-case isn’t described, it would be straightforward to do so.