Connecting Scripts

Definitions

Connecting Scripts refers to making one script read one input from the output of another script.

For example, a script that contains a synth *f* that adds reverberation may read its audio input from another synth *s* that produces an audio output. Thus the reverb effect of a synth in script *f* is added to the audio output of the synth in script *s*. The script that outputs the source will be called here a "writer" because it writes its output to a signal while the script that reads the output of the writer is called a "reader".

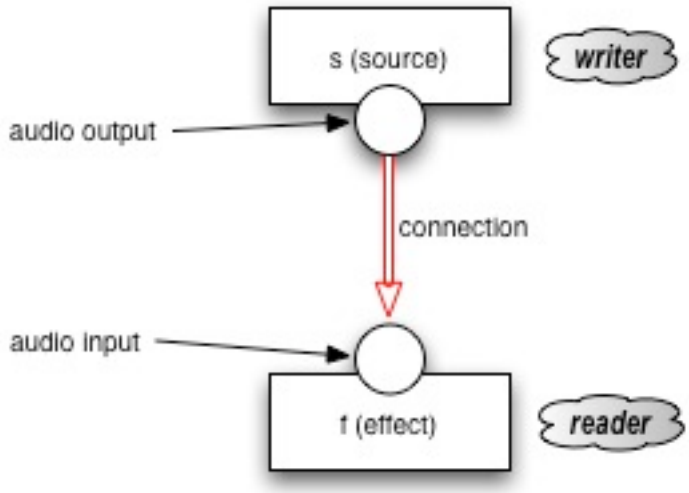


*Fig. 1: A script s is connected to a script f. The output of s is sent to an input of f. s is the **writer** because it writes on a bus (the connection) and r is the **reader** because it reads from that bus.*

There are four possible types of connections depending to the number of script-outputs and the number of script-inputs that are involved:

1. 1 to 1: 1 output from 1 script is sent to one input of another script
2. n to 1: several outputs from several scripts are sent to 1 input of another script
3. 1 to n: 1 output from 1 script is sent to several inputs of several scripts
4. n to n: several inputs from several scripts are sent to several inputs of other scripts, where each output is sending to several scripts and each input is receiving from several scripts.

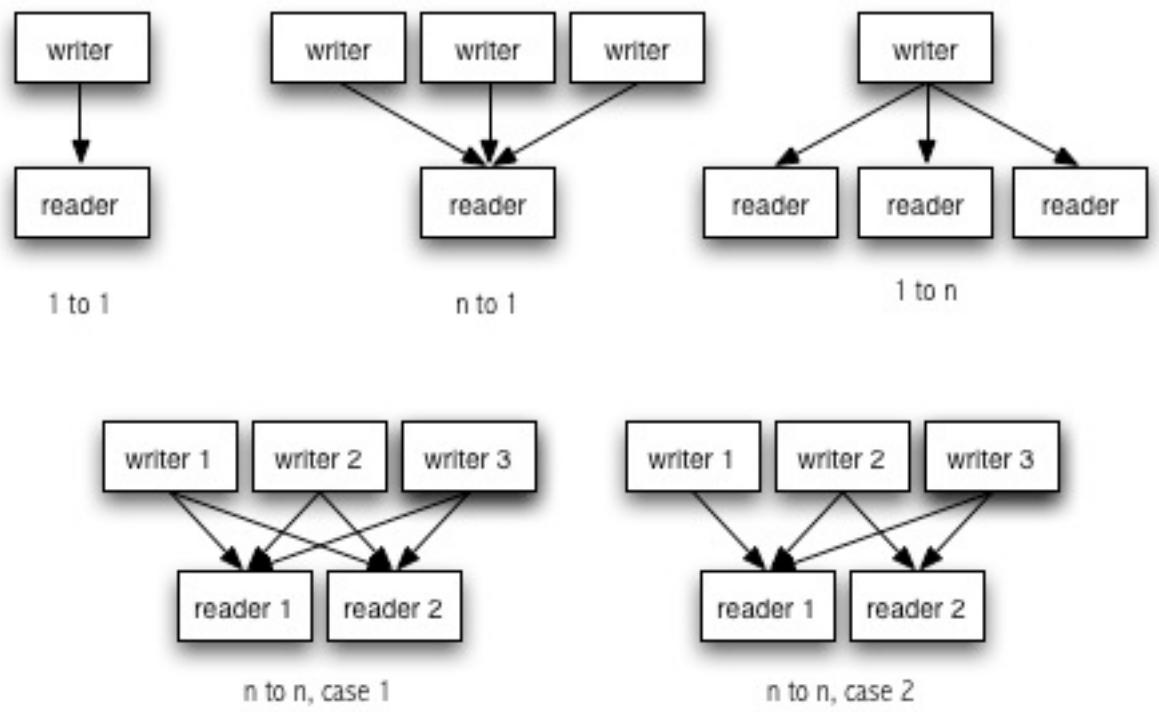


Fig. 2: 4 types of connection-sharing between synths.
In n-to-n case 1 all readers in the group share exactly the same set of writers. In n-to-n case 2, reader 1 reads from writers 1, 2 and 3 while reader 2 reads only from writer 2 and 3. Therefore in case 2 reader 1 and reader 2 cannot read from the same bus (see below).

The n-to-n connection (type number 4 above) is the most general type of connection, that is, the other 3 cases can be seen as subcases of n-to-n, which arise when either the senders (writers) or the receivers (readers) or both are only 1. As shown in Figure 2, the fourth type of connection has two subtypes, depending on whether all readers are sharing all outputs of all writers or not. This is important for implementation, because in order for a writer to send its input to a reader, it must write the output to a bus which is connected to the input of a reader. An output can only write to one bus at a time and an input can only read from one bus at a time. Therefore, the more general case case 2 of n-to-n in Figure 2, where one reader may share only a subset of the writers of another reader, requires the copying of the signal from the bus of the output to a different bus of the input, as shown in Figure 3:

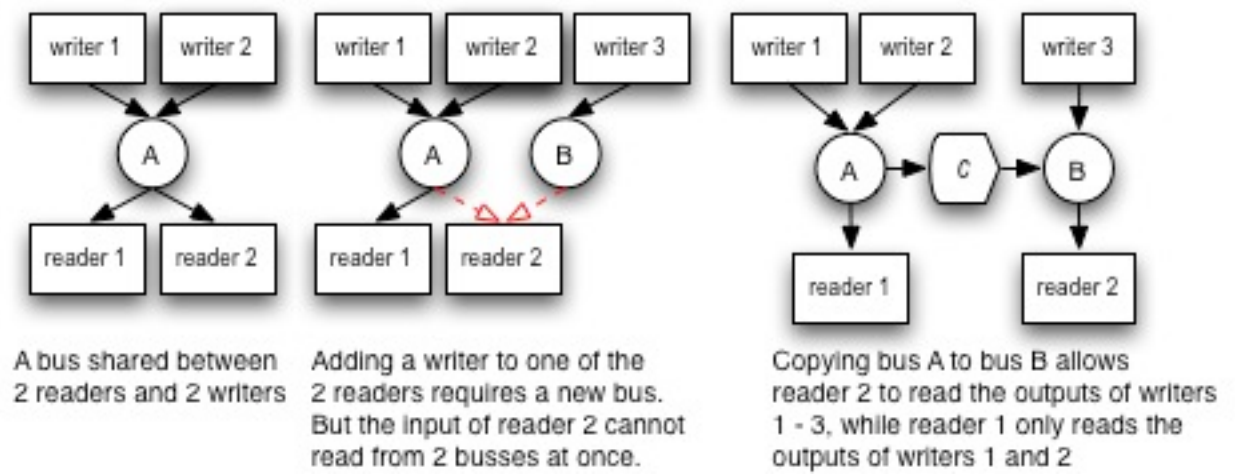
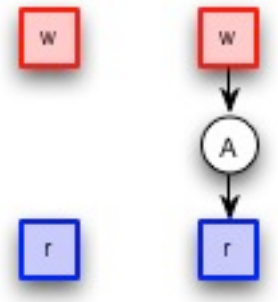


Fig. 3: Copying bus contents for n-to-n connections of type 2

Basic Principle of Link Creation

Following illustration of the role of bus-copying synths in n-to-n connections serves to explain the rationale of the general algorithms for adding and removing links, which are described in the next sections.

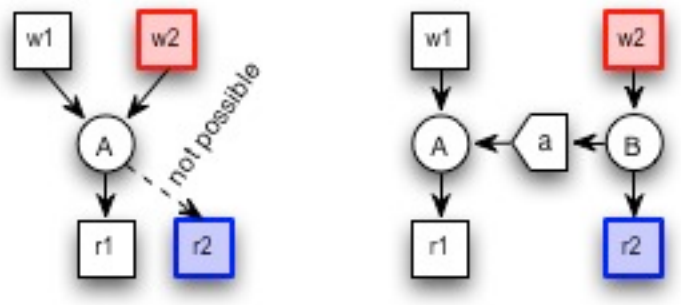
When the writers output is not yet connected and the readers input is not yet connected, the solution is simple. Create a bus and connect the writer's output and the readers input to it:



However, as explained above, if a reader r2 is to be connected to a writer w2 who is already writing to a bus, and moreover that bus also receives input from another writer w1, then it is not possible to add a link between r2 and the bus A that w1 and w2 are already writing to, because in that case r2 would receive input both from w1 and w2, whereas the intention is to create a link between r2 and w2 only. Therefore, following steps are necessary:

1. Create a new bus B that will receive the output of w2 only and from which r2 can read.
2. Move the output link of w2 from bus A to bus B (w2 will now output to bus B instead of bus A).

3. Connect the input of r2 to bus B so that it will read the output from w2 (only).
4. Copy the signal of bus B to bus A so that r1 may still receive the input both of w1 (which is directly linked to bus A) and of w2 (which is now linked to bus B).



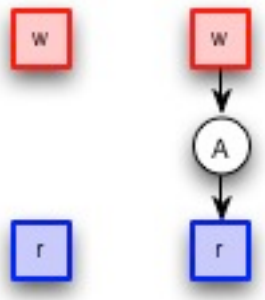
The algorithms for adding and removing connections between synths that are described in the following section are constructed based on the above principle, by distinguishing all possible types of configurations between writers and readers with links to busses.

Cases and Algorithms for Creating a Connection between a Writer *w* and a Reader *r*

The choice of the algorithm to be applied depends on the types of interconnections that already exist at the output of w and the input of r. The following cases exist:

1. w has no output bus and r has no input bus yet.

Create a new bus and make w write to it and r read from it.

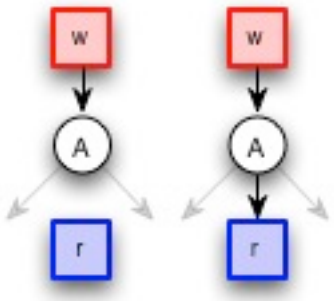


2. w has an output bus and r has no input bus

Here there are two subcases:

2a. The output bus of w has no further inputs (writers)

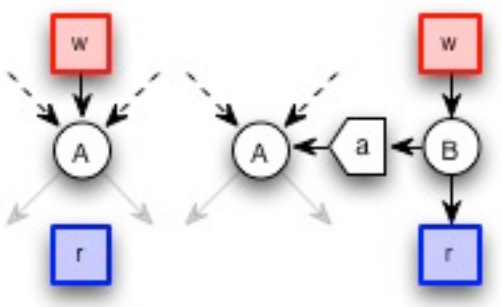
Connect r to A.



(the gray connection lines leading from A indicate that A may write to readers other than r but that this does not matter for the connection of w to r.)

2b. The output bus A of w has further inputs (writers)

1. Move w's output to a new bus B so that r may receive it separately.
2. Connect r to B.
3. Copy B to A so that other readers of A may keep receiving w's output.

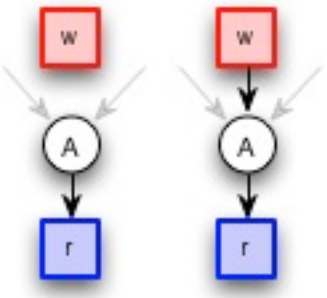


3. r has an input bus and w has no output bus

Here there are 2 subcases:

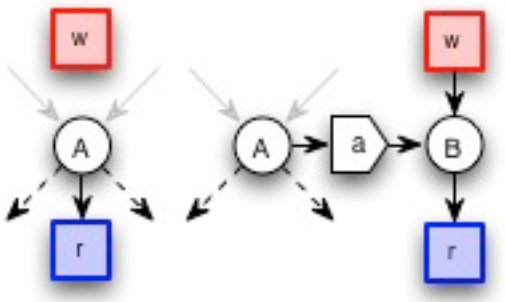
3a. A has no other readers than r

Connect w to A.



3b. A has other readers than r

1. Move r to a new bus B so that it may receive w's output separately from the other readers of A.
2. Connect w to B.
3. Copy A to B so that r may keep receiving the output of other writers than w.

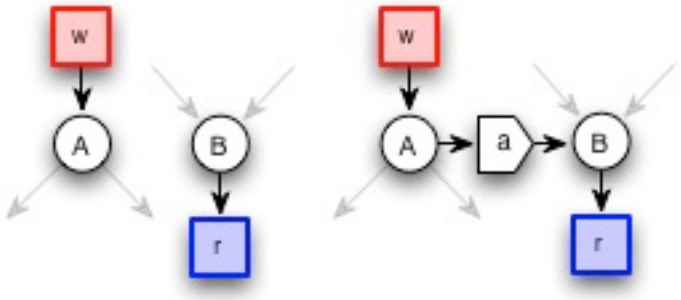


4. w is already writing to a bus A and r is already reading from a bus B

Note that this implies that already other readers are reading from A and other writers are writing to B. There are 4 subcases:

4a. A has no other writers than w and B has no other readers than r

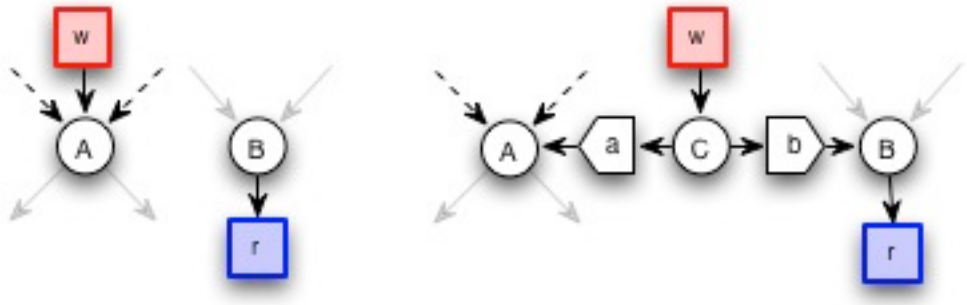
Copy signal of A to B with a connecting synth a, thereby adding w to the writers of B and therefore to r. (keep A and B separate because they have different readers and writers respectively).



4b. A has other writers than w and B has no other readers than r

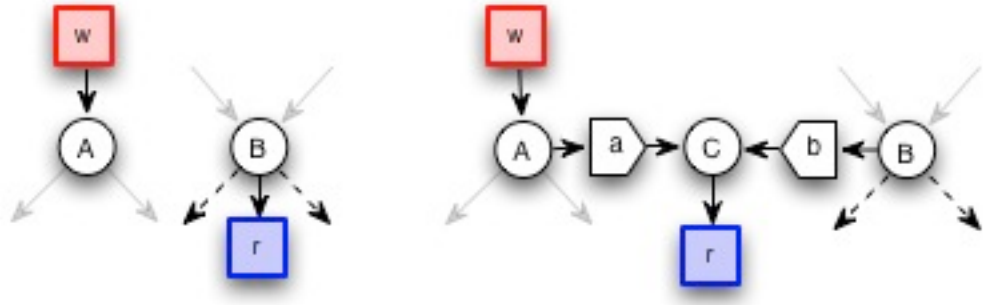
1. Move w to a new bus C so as to be able to send its output to r separately from the other writers writing to A.

2. Copy the signal of C to A so that the readers of A keep receiving the output of w.
3. Copy the signal of C to B, thereby adding it to the writers read by r.



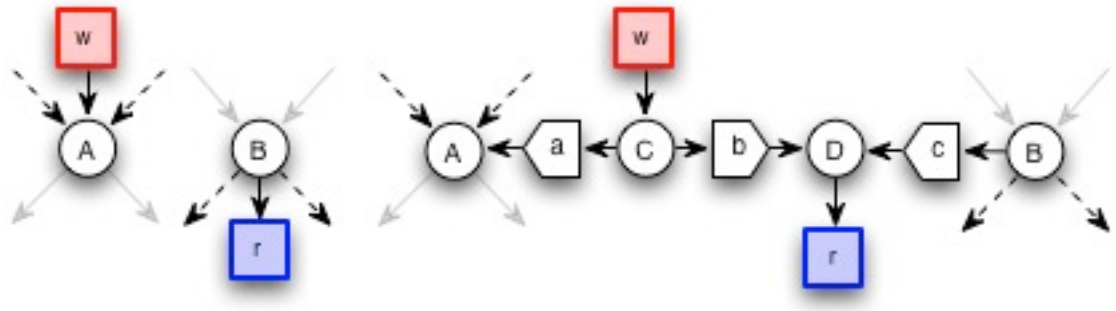
4c. A has no writers other than w and B has readers other than r

1. Move r to a new bus C so as to be able to connect w to r without the other writers of B receiving w.
2. Copy the signal of B to C so that r keeps receiving the outputs of the writers of B.
3. Copy the signal of A to C, thereby connecting w to r.



4d. Both A has writers other than w and B has readers other than r

1. Move w to separate bus C so that it can send its output to r separately from other writers.
2. Move r to separate bus D so that it can read w separately from other readers.
3. Copy C to A so that the readers of A keep reading w.
4. Copy C to D, thereby connecting it to r.
5. Copy B to D, so that r keeps receiving the writers of B.



[unless all readers of B have the same writers ???]

Generalization of above cases to rules, and features of the resulting graphs.

The basic rules for adding connections between synths that are already connected are:

A. If an input bus A of a reader r has more than one readers, then to add a writer w to r one must:
1. Create a new bus B, that will connect the output of w to the input of r
2. Remove r from the outputs of A
3. Copy the contents of A to B

B Conversely, if an output bus A of a writer w has more than one writers, then to add a reader r to w one must:
1. Create a new bus B that will connect the output of w to the input of r.
2. Remove w from the inputs of A.
3. Copy the contents of B to A.

A direct consequence of the above rules is that whatever the number and complexity of interconnections added, the resulting graph will always have busses interconnected in one of the following two ways:
(a) The bus has one writer only and one or more readers
or:
(b) The bus has one or more writers and one reader only.

That is, there will never be a bus that has both more than one writers and more than one readers.
Moreover, and more importantly, **a bus interconnecting synth will always copy a signal from a bus that is only receiving input from one writer to a bus that is only writing output to one writer**. This is characteristic is important for the algorithm that removes bus interconnecting synths when disconnecting readers from writers.

Removing Connections

The cases described above for adding a connection show an initial configuration ***i*** before the connection and a final configuration ***f*** that results after the connection has been added. When removing a connection, the reverse process should take place, that is for each case described above, one should move from the state ***f*** (connected) to its corresponding state ***i*** (disconnected). The following algorithm does the job:

1 Find the last node in the path from the writer to the reader which still has only one writer in its input

Meaning specifically: Traverse the graph of interconnections starting from the writer and going towards the reader, and stopping at either one of the following conditions:

- 1.1 the node which was reached has more than 1 writers.
- 1.2 the node which was reached is identical to the reader.

2 Let n be the node found as a result of the above traversal and n-1 be the node previous to n in the path of the traversal. Then:

2.1 Remove n-1 from the inputs of n.

- 2.1.1 If n is an input parameter, then set the input of n to nil (which mutes the input of n)

2.2 Remove n from the outputs of n-1.

- 2.2.1 If n-1 is an output parameter, then set the outpout of n-1 to nil (which mutes the output of n-1)

3 Remove redundant output paths from the output of the writer.

This consists of:

3.1 If the output bus o of the writer w is not nil, then

3.1.1 If o has no readers,

- 3.1.1.1 Set the output of w to nil (which mutes the output of w)
- 3.1.1.2 Free o.

3.1.2 Else if o has only one reader, and this reader is a BusLink l (bus interconnecting synth) then, let o2 be the output bus of l, and:

- 3.1.2.1 set the output of w to o2
- 3.1.2.2 add w to the writers of o2
- 3.1.2.3 stop l (if it is running)

4 Remove redundant input paths from the input of the reader.

This consists of:

4.1 If the input bus i of the reader r is not nil, then

4.1.1 If i has no writers,

- 4.1.1.1 Set input of r to nil (which mutes the input of r)
- 4.1.1.2 Free i.

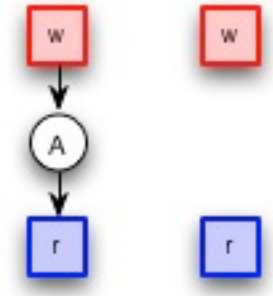
4.1.2 Else if i has only one writer, and this reader is a BusLink l (bus interconnecting synth) then, let i2 be the input bus of l, and:

- 4.1.2.1 set the input of r to i2
- 4.1.2.2 add r to the readers of i2
- 4.1.2.3 stop l (if it is running)

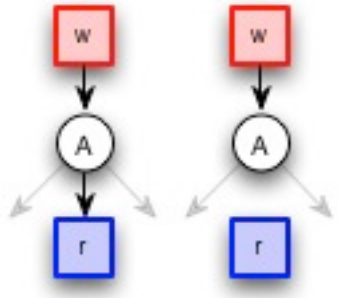
======== FOLLOWING ALGORITHM HAS FAILED - KEPT HERE FOR THE RECORD ONLY =====

The algorithms for removing connections follow the same rules as those for adding connections with respect to separating busses so as to maintain the right flow of signals between readers and writers. There are 4 cases:

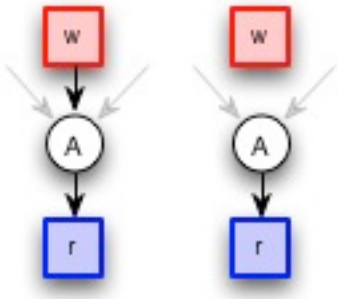
1. A has only one reader and one writer
Remove and deallocate A. Set the output bus number of w to the default output and the input bus number of r to the default input.



2. A has only one writer and more than one readers
Remove r from A. Set the input bus number of r to the default input.



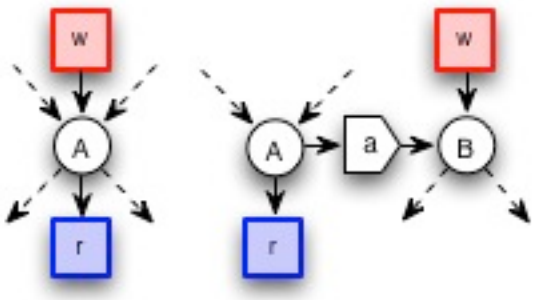
3. A has more than one writer and only one reader
Remove w from A. Set the output bus number of w to the default output.



4. A has more than one writer and/or more than one reader

1. Remove w from A and onto a new bus B.
2. Move all readers of A except r to B so that they keep reading w.
3. Copy A to B so that the readers of B keep reading the rest of the writers of A, plus w.

Now r has been left reading A but without w. r still receives output from all other previously connected writers to A. All other previously connected readers of A still receive the output from both the other writers and from w.



Rewriting the above rules for implementation:

Top method: **OutputParameter:removeReader**
1. The reader (method: **Parameter:removeWriter**)
Check the reader's input to see if the writer is the only writer of this reader.
If yes, then:
(1a) (*on the input bus side of the reader parameter:*)
Call **LinkedBus:removeReader**, on the input bus.
This does the following:
Removes the reader from its readers.
Checks if it has other readers.
If it has other readers (case 2 above), it does nothing
If it has no other readers (case 1 above):
(1a1) It sets the writer's output to nil (Method **OutputParameter:output_**):
In response, the OutputParameter mutes its output bus.
(1a2) Release (close) (Method **LinkedBus:free**):
It deallocates itself and stops monitoring the server for automatic reallocation.
(1b) (*on the reader parameter side:*)
After the work has been done on this reader's input,
set this reader's input to nil. (Method **Parameter:input_**)
This mutes the parameters input (kr: unmap, ar: set to mute input bus).
If not, it leave the input bus as is.
(Note: this covers cases 1 and 2 above.)

2. The writer (continuing method **OutputParameter:removeReader**)

NEEDS REDOING!!!!
(a) (*on the output bus side of the writer parameter*):
If the writer has not been removed from the writers of the output bus through step 1a1 above,
then Remove the writer from the writers of the output bus.
(b) (*on the side of the writer parameter itself*)
check if the reader is the only reader:
If yes (case 3 above), then:
Set writer's output bus to nil (Method **OutputParameter:output**_):
This mutes the writer's output (sets it to the ar or kr mute output bus)
If not, then:
Create BusLink copying old bus to new LinkBus.
Move all ***other*** readers to the new bus.
Set this writer's output to new bus.

3. Notifications of scripts for updates of dependants (guis etc)
(continuing method **OutputParameter:removeReader**)
(a) Notify writer's script that its readers are changed
(b) Notify reader's script that its writers are changed

======== END OF FAILED ALGORITHM =====

Removing Superfluous Link-Synths and Merging Busses

When ???

Computation Order ([Order-of-execution])

In order for the reader to be able to read the signal output by the writer, the writing synth must be placed before the reading synth in the order of execution (see *[Order-of-execution])*. To ensure this, in the present implementation, the writer and the reader are placed in separate groups so that the group of the writer precedes the group of the reader in the *[Order-of-execution]* of the synth engine. This ensures that the reader can read the output of the writer. Also if either the writer or the reader already have writers or readers respectively, these will be moved to preceding or following groups so that the order-of-execution is always the correct one for all synths involved. The details of this mechanism are explained below. The steps for creating links and keeping the synth graph sorted in the correct order-of-execution are the following:

1. Check if the new connection will create a cycle. If yes, reject the connection (possibly posting a message).
2. Place the writer and the reader in appropriate groups so that the writer is in a group previous to the reader.

3. If needed, move the writers of the writer and/or the readers of the reader to other groups so that the order of execution stays correct.

Sortability of a synth-graph, cycles

In order for all connections between synths in a given work session to work, synths must be sorted so that for every synth w whose output is read by another synth r, w is placed before r in the order of execution. The set of connected synths is called synth graph. Finding out the correct order of execution of the synths in the graph according to the sorting condition just explained is called sorting the synth graph. It is always possible to sort a graph in the above manner, as long as it does not contain cycles, that is, as long as the chain of connected outputs to inputs does not lead back from a reader r to a writer w that writes to the reader, so that the reader both reads from and writes to the writer:

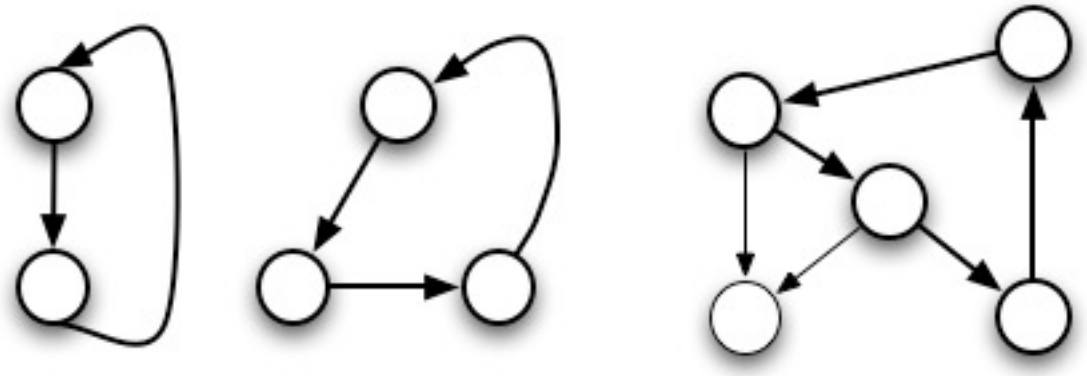


Fig n: Examples of synth graphs containing cycles. The cycles are highlighted with thicker lines

Cycles will not be allowed in the present system at the moment (Their implementation is possible but as this is a case that is used only very seldom, it will not be realized here at this stage). If there are no cycles, the graph will always be sortable. Therefore, a check is introduced before the creation of each link to make sure that the link will not create a cycle.

Keeping the synth-graph sorted by use of groups

In order to keep the synths of the interconnected scripts arranged in the right order-of-execution, each synth is assigned to a group out of a number of groups that always run in a fixed order. Such synths are always created ***at the head*** of the group to which they ar allocated (in contrast to bus interconnecting synths that are allocated at the tail of the group). These groups form as it were a kind of "tier" system, and each synth is assigned to one tier-group so that all its readers are on one of the following groups and all of its writers are on one of the preceding groups:

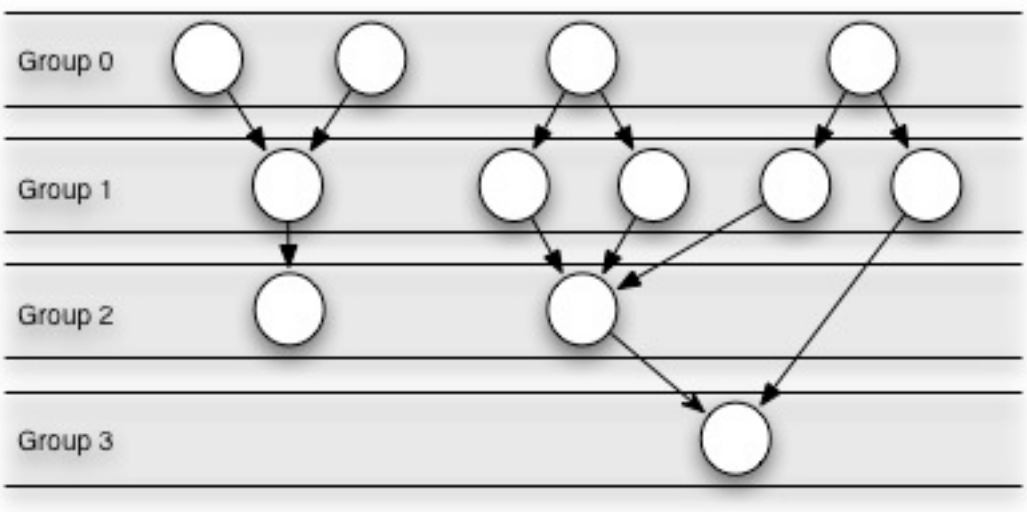


Fig. n: Arranging the order-of-computation of synths by assigning to ordered groups

The reason for using groups as targets of the synths rather than the previous or next synth in the graph of synths as depicted in Figure 3, is that groups provide a fixed framework that exists independently of the synths. This framework allows one to start and stop each synth independently of the other interconnected synths in the graph, while always keeping the correct order. To do this without groups would be much more complicated.

The "tier groups" are saved as array in the session to which the scripts belong, in order from earliest to latest. They are started as soon as their session is loaded and remain running as long as their session is open. A script can allocate itself to a group by knowing its index in the array of groups in its session. It accesses that group by its index in the array and stores it as target in its environment.

Group assignment when adding a connection

Given a script that is either a writer *w* or a reader *r* to be connected to a bus *b*, assign to it a group tier-number *g* and adjust the group tier-numbers of all readers so that the order-of-computation is sorted (the tier-number *g* of each group corresponds to its order of computation relative to the other groups):

When connecting the output of a writer *w* to a bus *b*:

1. If the writer does not have a group tier-number, assign *g* to 0. (Note that since w did not have a group, it had no writers, so no need to check its inputs).
2. Check the groups of the readers of *b*: If a reader *r* of *b* has a group whose tier-number is not greater than *g*, then assign the group of *r* to group number g+1, and proceed to check all the readers of r, so that their group numbers are greater than that of r, and then the readers of the readers and so on. When doing these adjustments, if any reader r is running, then its synth must be moved to the new group assigned.

Group reassignment when removing a connection

When removing the connection of a script x to a bus b:

1. If the script of *x* has no other inputs or outputs connected to busses, set its target to the server of the session of the script (this defaults the target to the default group of that server).

Following may not be implemented yet:
[**TODO**: *We need a mechanism for freeing groups that are not needed and moving scripts up to smaller tier-numbers when links are removed:*

2. If the script of x has no other inputs connected to busses and its group is not 0, set its group to 0 and move the groups of the scripts connected to its output to the least number above 0 that will preserve the right order-of-execution.]

Group assignment of bus-interconnecting synths

All interconnecting synths, which copy the signal from one bus to another as in the connection algorithms above, are allocated ***at the tail*** of the group with the greatest tier number amongst the groups of the scripts writing to the bus that the interconnecting synth is copying from.

How are groups assigned to scripts?

When a script is linked to another one, it is assigned a group tier-number, which corresponds to the index of a group in the array of groups stored in variable groups in the session that this script belongs to. This number is saved in environment variable ~groupNumber in the scripts environment (envir). The procedure for setting a group as target for a scripts synth nodes is:

1. Assign a group tier-number to the environment variable ~groupNumber in scripts envir.
2. Add the script to the array of linked scripts in Session variable linkedScripts.
3. If the server of the session is running, then the ~target variable of the script is set to that group from the sessions group array that is found at index ~groupNumber. Otherwise, it is
4. If the sessions group array has a smaller size than that required by ~groupNumber, then:
    4a. Update variable numGroups of session to reflect the required number of groups.
    4b. If the server is running, create and add new groups to the groups variable of the session until it has reached the appropriate size,
    4c. Obtain the appropriate the group for the script.

5. Whenever the server boots or the command-period is pressed, the session creates new groups and updates the ~target variables of all scripts contained in linkedScritps based on their ~groupNumber.

When are groups created and freed?

The number of tier-groups needed by a session are saved in instance variable numGroups. The tier-groups of a session are created as soon as the server of that session boots. If the server of the group is already running when the session is created, then the groups are created immediately. The Main-stop method of SuperCollider is programmed to free all groups when the key command-. (Command-Period) is pressed. Therefore, a Session will also re-load its groups every time that command-. is pressed.

Every time that new groups are created, session assigns to those scripts that are linked the group that corresponds to their tier-number as target. This number is stored in ~groupNumber.

When a new group needs to be added to the array of tier-groups because a linking of script requires the addition of a new tier in the order-of-execution, the group is immediately created and assigned to the scripts that require it. Furthermore, the number of groups (numGroups) is updated with the actual number of groups. Conversely, when the number of groups lessens because of removed links, the free (empty) groups are freed and numGroups is updated accordingly.

When the Server boots, session waits for 1 second before allocating its groups, because under certain circumstances at boot time removing that delay may lead to failure in creating new groups, due to the implementation of method doWhenBooted in the standard SC library.

When a session closes, it frees all its groups.

When are busses allocated and deallocated?

Whenever the creation of a link between two scripts requires the creation of a new bus, a new instance of LinkedBus is created, and a bus number is allocated from the bus allocator of the server of the session of that script. This is independent of whether the server of the session is currently running or not. The bus number of the LinkedBus instance is given to all those parameters of scripts that write to the bus or that read from it.

When a server quits, its bus allocators are voided and new bus allocators are created. These new allocators remain valid throughout the next boot of the server and until the next quit of the server. For that reaon, when the server of a session quits, the LinkedBus instances of all scripts contained in variable linkedScripts of that session are all reallocated new bus numbers obtained from the server. These numbers are allocated to the

LinkedBus instances and to those parameters of scripts that write to them or read from them.

How are synth parameters of scripts set or mapped to the appropriate bus index numbers?

An ar parameter is linked to a bus by setting the value of the parameter to the index number of the bus. This can be done at any moment, independently of whether the

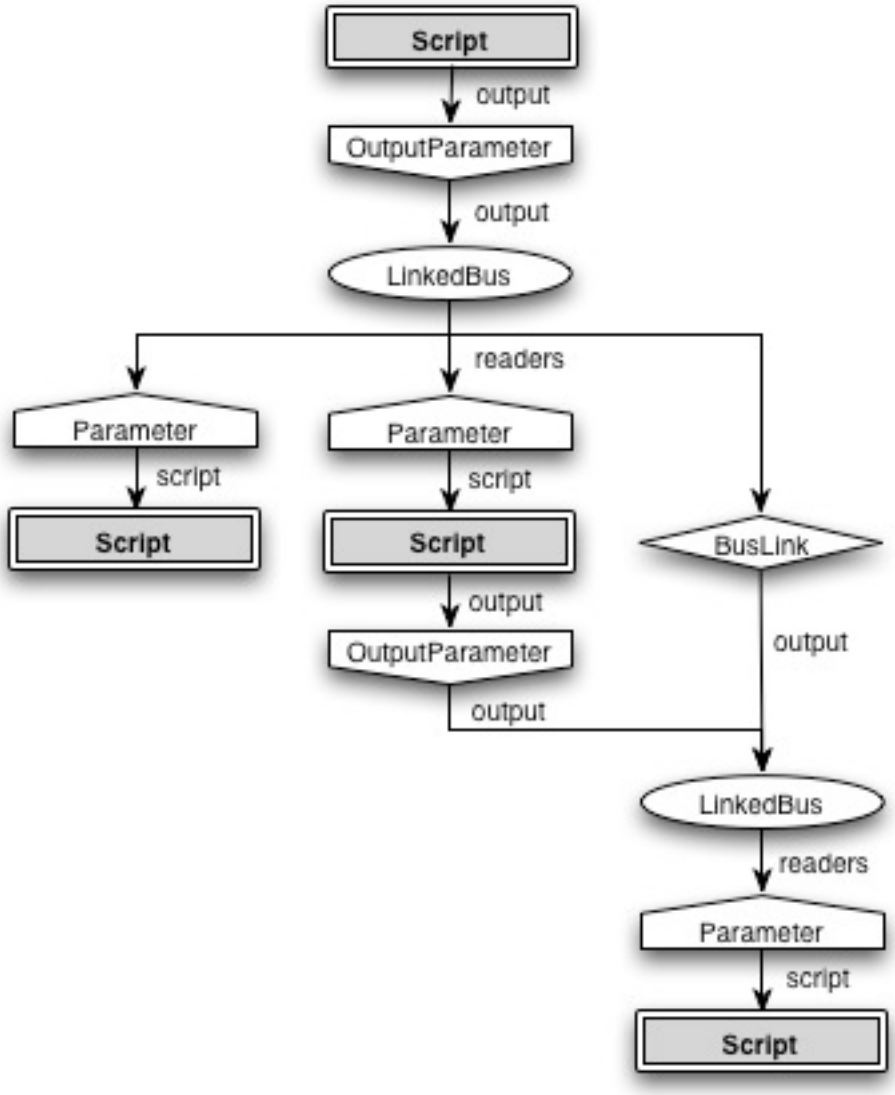
A kr parameter is linked to a bus by mapping the parameter to the index of the bus.

Details:

The setting of the ar param

Chain of objects implementing a link between two scripts

There are at least 3 instances of different classes between a scripts output and another script that reads that output. The structure of a connection between a scripts output and the next scripts input are is as follows:



1. A script has one output (instance variable *output*), which is an obect of kind Output-Parameter (either a KrOutput or an ArOutput).
2. An OutputParameter has one output (instance variable output) which is an instance of LinkedBus
3. The signal of a LinkedBus can be read by any number of readers where each reader is either
    3a of kind Parameter or
    3b of kind BusLink
4. 4a A Parameter p inputs signals from a Linked bus and outputs them to a script. The script is contained in the *script* instance variable of parameter p.
    4b A BusLink *b* outputs signals outputs its signal to a LinkedBus that is contained in the instance variable *output* of *b*.
5. A script receives inputs via one or more instances of kind Parameter stored in an array in instance variable parameters.

Roles of Objects in a Link between two Scripts

1. The ***Script*** holds the instances of parameters in its instance variable *parameters*. These include all its inputs and its output. Additionally, the output is stored in the instance variable output. These instance variables are used for access when creating the gui for the script as well as when linking scripts.

2. The OutputParameter stored in instance variable output of script is responsible for creating the drag-source view for linking the output to any parameter input. As Parameter it is also responsible for creating the parameter view (drag-sink label,

3. The LinkedBus reserves and

4. (

5. The Parameter instances that form inputs of a script create guis for direct user

Checking if a link can be created between a writer and a reader

When the user requests a link to be created between a writer w and a reader r, it must first be checked if a valid link can be created. There are 3 checks to perform:

1. Test if the type of the writer is compatible with the type of the reader (if the writer has audio-rate output, then the reader must accept audio-rate input).

2. Test whether a link between w and r already exists. If yes, reject the request: Do not create twice a link between the same writer and the same reader.

3. Test whether creating the link would result in the creation of a cycle. For example: if the request is to create a link between w and r and r is linked as writer to another reader r2 and r2 is linked as writter to w, then connecting w to r would create a cycle in the chain of write-read links. This would make it impossible to sort the chain of w - r - r2 in an order of computation, and therefore the link between w and r is not acceptable in this case.

Implementation of checking if a requested link already exists

When a link is requested between a writing parameter *w* and a reading parameter *r*, it must first be checked if that link already exists, in order to avoid adding the same link twice. Implementation:

When a link is requested between a writing parameter *w* and a reading parameter *r*, it must first be checked if that link already exists, in order to avoid adding the same link twice. Implementation starting from method Parameter-canLinkTo:

```
canLinkTo { | writer |
        /* Request from drag-sink or other to link a writer to my input.
        Test if writer is of right kind, is not already linked to me, and
        would not create cycles if linked */
        ^writer.isKindOf(this.acceptableWriterClass) and:
                { writer.doesNotIncludeReader(this) } and:
                { script.containsCycle(writer.script).not }
}
```

Implementation of doesNotIncludeReader:

1. Reader r asks writer w to confirm that it does not already contain a link to r:
writer.***doesNotIncludeReader***(this)
2. The writer is an OutputParameter. Its method *doesNotIncludeReader* is:

```
        ^output.isNil or: {
                output.readers.detect(_.includesReader(reader)).isNil
        }
```

Meaning it wil check the readers of its output. Its output is a LinkedBus and its readers are either

2a Parameter instances or
2b BusLink instances.
So the instances of readers here respond to method includesReader depending on their class as follows:

3. *includesReader* check by Parameter and BusLink instances
3a Parameter instances return true to includesReader(reader) if reader === this
3b BusLink instances return true to includesReader(reader) if one of the *readers of its output*
(a LinkedBus) returns true to includeReader(reader)

```
        output.readers.detect(_.includesReader(reader)).notNil
```

Implementation of checking for cycles

```
Script:containsCycle { | writerScript |
        // Check if i am trying to write to myself directly or via my outputs
        ^writerScript === this or:
        { output.notNil } and:
        // output: outputParameter, its output: LinkedBus, its readers: params or links
        { output.containsCycle(writerScript) }
}
Parameter:containsCycle { | argScript |
        ^output.notNil and:
        { output.readers.detect(_.containsCycle(argScript)).notNil }
}
```

Implementation of creating a new link between a writer and a reader

The request for creation of a new link between a writer and a reader will usually be issued as a result of the user dragging the drag-source view of an OutputParameter of a script gui to the drag-sink view of a Parameter in another scripts gui. It will thus be issued from the reader parameter as a request to be added as reader to the OutputParameter that is currently the dragged object:

```
SCView.currentDrag.addReader(this);
```

Implementation of choice of interconnection type

See further notes in ScrtipLinkingAlgorithm.rtf

Implementation of group allocation

Conclusion

The objective of Lilt is to provide a clean implementation of the above, easily useable at the level of coding, as well as a graphical user interface that lets one create, remove, view and manage connections with simple drag-and drop operations combined with a few keystrokes. The design of the interface is explained in: [Connecting Scripts.help.rtf]. Finally, the configurations created via the gui are saved along with the session as script, and can be re-created when the session is loaded.