The Integra protocol: a feasibility study

Jamie Bullock
Birmingham Conservatoire, UK

August 16, 2006

Abstract

This document discusses a series of tests conducted in order to establish the feasibility of the Integra protocol. In particular this document aims to test to what extent the protocol is "technology independent" and " can be implemented on any platform and software supporting OSC"[1]. A simple patch involving a fixed number of Integra modules was created in three existing environments for audio. Conclusions drawn about the implementation of the protocol in each environment.

1 Protocol description

The Integra protocol is based on the concept of defining module interfaces that encapsulate a higher level of functionality than that provided natively by environments currently used for 'music with electronics'. Communication between module instances is achieved by passing lists containing an address string and an arbitrary number of parameters exclusively from one module to another. The strings represent OSC addresses, and are tested against lists of existing strings in the recepient module to establish what the consequent action should be. If the address string is valid, it can either cause the evaluation of a function, and/or a variable assignment. If parameters are provided, they are either used in assignment, or passed directly to functions.

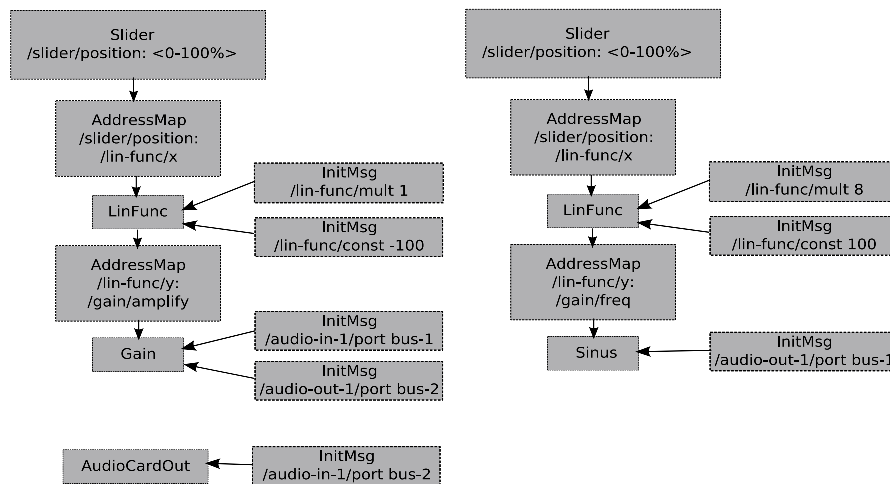
Therefore as a minimum, any environment wishing to support the Integra must support the exclusive passing of arbitrary argument vectors directly between modules, comparisons between strings, and function evaluation. By exclusive message passing, it is meant that a message send from a sender to a receiver must not go to any other modules.

Implied throughout the protocol definition is the notion of inheritance between modules. This means that if 'module A' inherits 'module B', 'module

A' gains all of the functionality frm 'module B'. Attempts have been made to implement this in these tests.

2 The task

The task is to create an implementation of the following network in three contrasting environments by creating and using Integra modules.



3 Evaluation criteria

Each environment will be evaluated on its ability to support the namespace, and implement module inheritance. Important factors are the amount of user input required to create the modules, the amount of user input required in arranging the modules in a patch and the computational efficiency of the implementation.

4 The environments

4.1 PD

4.1.1 Integra modules are created as follows:

- Create instances of PD objects with arguments
- Connect objects with patch chords
- Save modules as re-usable 'abstractions'
- Connect modules (abstractions) using patch chords
- The DSP graph is built entirely at runtime using patches

4.1.2 Classes

- All objects are instances of classes
- It is possible to create new classes either in C or in PD
- There is no builtin inheritance
- Inheritance can be faked by embedding abstractions
- Methods are called via message passing
- First inlet is hot (causes output)
- Output is right to left

4.1.3 OSC namespace

- OSC support is via an implementation of CNMAT's original OSCtools
- Basic routing and wildcards are supported
- There is some magic (it is not necessary to specify type tags)

4.1.4 Routing

- Routing is achieved by patch chord connections, bus objects or OSC objects
- Order of execution is controlled by trigger objects.

4.2 SuperCollider

4.2.1 Integra modules are created as follows

- Synthesiser definitions (SynthDefs) are written in the Supercollider language by connecting unit generators (UGens)
- the client compiles SynthDefs into byte-code
- Conditional expressions in synths are evaluated at synthdef compile time
- the byte-code is read by the server at synth load time and cannot be changed after load
- Synths can be controlled directly using Supercollider server's OSC namespace, or 'indirectly' using the sclang client
- Synth instantiation and module inheritance can be controlled and exploited by writing Classes

4.2.2 Classes

- Classes are objects, instances of (Class)
- They are interpreted by the client (language)
- The language has OOP support
- It is therefore possible to create a hierarchy of modules, which inherit functionality from each other

4.2.3 OSC namespace

- It is possible have a user-defined namespace in the client
- Build in OSC routing support is basic, but can easily be extended
- The namespace used by the server is fixed
- The server can be controlled via OSC including adding Synths, changing the graph, changing routings.

4.2.4 Routing

- Order of execution is controlled explicitly by node allocation
- By default nodes order follows synth instantiation order
- Audio and control data routing is achieved via global busses

4.3 Csound

4.3.1 Integra modules are created as follows:

- Synthesis modules are created as Csound instruments in a CSound .orc (or .csd) file
- Synthesis modules are instantiated using a .sco (or .csd) file
- Language features are not separated from synthesis, so it is possible to have conditional expressions that are evaluated every performance pass.
- The Csound .orc file is compiled by the csound interpreter, and instr blocks are added to the graph in assending instr order.

4.3.2 Classes

- Csound is not object oriented, however, subinstr could be used to mimick a class-like behaviour
- The subinst technology is not mature, for example, it is not strainghtforward to pass strings as parameters to subinstruments. This rules out running the namespace over csound messaging
- Instruments can be instantiated arbitrarily in the score

4.3.3 OSC namespace

- Csound has basic OSC support, it can send and receive char, double, float, 64-bit int, 32-bit int and string
- Csound uses the liblo library, so Csound's OSC functionality could easily be extended
- It is possible to instantiate an OSC 'listener' inside an instr block, but its handle will be reinitialised if the instr is reinstantiated

4.3.4 Routing

- Order of DSP execution is determined by instrument number
- Basic routing (global) can be achieved using global variables/
- Audio and control data Routing between instruments can be performed using the zak patching system.
- Strings can be passed from the score, or using OSC
- Support for string passing between instruments is very basic

5 Results

5.1 PD

In PD it was possible to assemble a complete set of Integra modules, and adhere completely to the Integra protocol. All objects in the patch are Integra modules, with the exception of the bus definition subpatch. There were some difficulties in making the audio routing dynamic, it was therefore necessary to create a fixed number of named buses.

A new module 'InitMsg' was devised for these tests. This module sends out a predetermined OSC message when the module is instantiated - useful for setting initial settings. It was also used in the Csound and Supercollider implementations.

5.1.1 CPU load

[1]

- pd: 1.7-2.0
- pd-gui: 0.3-0.7

5.2 Csound

It was not possible to adhere to certain aspects of the Integra protocol due to limitations in the OSC implementation in Csound. In particular, it is not possible for Csound to respond to a an OSC message that has no arguments. It was therefore necessary to introduce an argument of '1' to addresses with

[1] With rapid parameter change on a 2.0Ghz Pentium M

a .get suffix. It was also necessary introduce a random number generator kludge because OSCsend will only send out a value if its first argument changes, and will always send a value if its first argument changes.

It was not possible to set the slider position at runtime due to a seeming bug the in FLsetVal opcode.

In Csound all OSC data is sent to defined UDP ports on defined hosts. Therefore OSC communication sent to one module on a particular port, will also go to all other modules listening to that port. Two alternative solutions to this were explored. The first was to give each module a unique prefix (modulename), see main.orc and main.sco. The second approach was to make each module listen on a different port. The approach of using separate ports proved extremely limited since each OSC 'listener' has a unique handle, meaning that if the listener is instantiated inside an isntrument block, the handle gets overwritted each time a new instrument instance is created.

These issues could be resolved by passing strings between instruments. This is currently possible, but only in a very limited way; using pop/push.

To save time .get methods were not provided for all modules, although this would have been perfectly feasible.

One drawback with Csound (despite the quality and flexibility of the widget set) is that graphical widgets are limited in the way they can be instantiated. For example, it is not possible to define a panel, and then later add widgets to itfrom inside instr blocks. In terms of implementing the Integra namespace, this means that there must always be an explicit OSCsend statement in the orchestra for every widget. In main2.orc, this has been done by adding a new instr block for each widget, this is to show the conceptual modularity of the implementation. A more concise solution is presented in main.orc. It is not currently possible to set the initial value of GUI sliders, hence the disparity in initial state between the Csound and PD/SUpercollider implementations. This is likely to change in future Csound releases.

The possibility of using module inheritance inside Csound was not explored. Csound has no OOP features, but it might be possible to fake inheritance as it has been done in the PD example; by using sub instruments.

5.2.1 CPU load

- Csound: 1.3-2.0

5.3 SC3

Supercollider's language side OSC functionality is similar to Csound's in that a 'listener' is established on a particular UDP port, and actions are associated with input addresses. OSC connections are therefore non-exclusive unless a unique address prefix is used for each module instance, or separate ports are used for each module. The approach taken in main.sc is to use unique module prefixes. However, an approach more akin to the Max/MSP or PD message passing style is to use string passing between modules. This is explored in main2.sc, and using this technique it was possible to implement the Integra protocol as it is defined by the specification.

Supercollider was the only environment where it was possible to properly implement a module inheritance model. This was achieved by writing new Supercollider classes. A generic module called IntegraModule was defined and inherited by all other modules, acting like a factory to associate module actions with OSC address strings. However, under its current definition, the Integra protocol has an implied multiple inheritance model. This is not directly compatible with many programming languages such as sclang that only support single inheritance. A slight deviation was made from the protocol with regard to Audio functionality. This is proposed as a replacement for the current audio module specification since it solves a number of problems.

A generic audio module (AudioIO) was defined and then inherited by any modules that require audio input or output. Again the module is factory-like, and generates addresses depending on howmany audio inputs or outputs are required by the inheriting module. For example if a module, Gain2 (a stereo amplifier), is created with the outPorts and inPorts arrays (inherited from AudioIO) each having 2 elements, then AudioIO would automatically create the addresses /audio-in/0/port, /audio-in/1/port, /audio-out/0/port and /audio-out/1/port for Gain2.

When scsynth (the server) starts, 127 buses are created automatically. These can be used for audio or control rate data, with the low order buses being used for the audio hardware I/O. In the Integra clases, the '/port' address was used to route between these buses. In IntegraClasses2.sc and main2.sc, message domain routing between modules is flexible, with the only caveat being that receiver modules must be instantiated before senders. This could possibly be avoided by using module proxies. It is also necessary to instantiate synthesis nodes in the correct order: receiving nodes first.

Graphical modules were implemented using SwingOSC. Currently there are limitations in this, which means that certain widget characteristics (size, position) cannot be changed after the widget has been drawn. This meant that it was necessary to place the sliders in separate windows to avoid having

them being drawn on top of each other. Another limitation was the granularity of the output from the sliders. This had noticeable 'steps', which made the sliders unsuitable for certain applications.

The SuperCollider implemenation took the longest time to produce, but is possibly the most scalable given the class inheritance and separation between classes, synthdefs and runtime code.

5.3.1 CPU load

- sclang: 0.3
- scsynth: 0.7
- java: 0.3 - 1.0

6 Overall

In both Csound and Supercollider, it was necessary to draw a window before drawing GUI widgets. It is therefore suggested that there is an IntegraWindow module in the protocol. It was also necessary in both Csound and Supercollider to define the absolute size and position of widgets relative to their parent window. Since it is possible to specify these in MaxMsp and PD also, it is recommended that this becomes part of the namespace.

All three of the current implementations make the assumption that only one parameter will be sent with the OSC messages. This is almost certainly not going to be the case as the specification evolves, so this needs to considered in any further work.

For all of the implentations, creating Integra modules required significantly more user input than creating equivalent implementations without the Integra protocol. However, this could be greatly eased through module templating and/or automated module builds. An idea of the 'effort' entailed in creating Integra modules can be gained from fig.1. Adding the Integra protocol to a version of a patch that contains the same functionality creates an additional 197 PD object instances, 222 lines of SuperCollider code or 122 lines of Csound code. In all three environments assuming that the Integra modules had alreay been created, the patches would have still involved more user input than simply creating native non-Integra patches.

There is also a message passing overhead, for a simple slider controlling a frequency parameter, without the Integra protocol, one message is passed, with the Integra protocol we have a message passed for each of the follow-

Table 1: Lines, characters and objects in patches

Environment	Lines	Chars	Objects
SC3 1	244	7903	N/A
SC3 2	255	7721	N/A
SC3 w/o Integra	22	578	N/A
Csound 1	136	3689	N/A
Csound 2	158	3536	N/A
Csound w/o Integra	14	337	N/A
PD	N/A	N/A	203
PD w/o Integra	N/A	N/A	6

ing: Slider, AddressMap, LinFunc, AddressMap, Sinus, a ratio of 4 Integra messages to 1 'environment native' message.

7 Conclusion

In these tests it has been shown that to a certain extent, the Integra protocol meets its aims in being "technology independent" and able to "be implemented on any platform and software supporting OSC"[1]. It was possible to create a simple patch following the Integra protocol in Pure Data and Supercollider. However, in Csound the limitation on string passing between 'instruments' meant that either each module needed a unique prefix, or each module required a unique UDP port. The latter of these options is unscalable in the implementation presented here. The module creation process was time consuming although it is anticipated that once models have been agreed for each environment, module creation times should improve. The message passing overhead is a much more serious problem. With GUI input, where only one element is likely to be active at one time this unlikely to be a problem if the object chain is not too deep. However, with rapidly changing multidimensional input (e.g. from sensors), it is anticipated that the introduction of the Integra 'layer' could cause CPU load problems.

One possible solution to these issues is to change the nature of the protocol. The protocol could define the response side only, with output addresses left undefined. This would alleviate the requirement for AddressMap modules between every module. It would also help reduce patch 'pollution' if the functionality provided by InitMsg could be inherited into modules, rather than distinct. This would mean that modules could have their initial state set 'internally' rather than by another module. This would reduce the number of modules required to create these implementations by more than half.

8 Further work

It would be useful to repeat these experiments in other environments, and make additional comparisons. It would be particularly interesting to attempt an implementation of the namespace in an environment designed for 'commercial' music such as Reaktor. The tests should also be extended to encompass more demanding synthesis tasks. Once the protocol has been finalised, a range of module templates or means to autogenerate modules should be developed.

9 References

1

Sundt, H. 2006, *Integra Developer Documentation, Integra website,*

Online

Available at http://www.integralive.org/integra-OSC/index.html (accessed 16th aug 2006)