COMPOSITION WITH SOUND WEB SERVICES AND WORKFLOWS

John ffitch, James Mitchell, Julian Padget
Department of Computer Science
University of Bath, United Kingdom

ABSTRACT

Sound synthesizer programs such as Csound, Max/MSP, PureData and Supercollider offer the composer a powerful stand-alone environment for the creation of sounds, instruments and complete performances, but sharing, publishing and collaborating are difficult for largely technical reasons, such as platform and configuration. The enforced isolation also necessarily leads to reinvention rather than development and refinement. Service-oriented architectures built on web services are an emerging technology that aims to help interoperability and increase flexibility by decoupling consumer and provider as long as the user is prepared to pay the price of more complex protocols and bear network latency. The potentially high computational load of some synthetic instruments, their sensitivity to the runtime environment and the possibly not unreasonable desire to maintain direct control over a delicate entity, makes the case for investigating the application of web services to the delivery of sound synthesis. Furthermore, the tools that have been developed for joining together web services, into so-called workflows, mimics the patch mechanisms in some synthesizer graphical user interfaces. We report upon the construction of a prototype system for the deployment of sound synthesis software as web services in which we used the Triana and Taverna workflow systems to combine such services. We conclude with an outline of how our related research into the semantic description of mathematical web services and matchmaking and brokerage of such services could be adapted for the domain of sound, opening the way to utilising the entire web as a source of synthetic instruments and computational power to generate music.

1. INTRODUCTION

The creation of electronic music by sound synthesis is hard work, and requires many skills. Debugging the audio program is even worse as many of the parameters are non-linear, and results can be unpredictable. Unlike traditional composition, in addition to deciding on the notes there is the problem of building the synthetic instruments, which is an abstract process. The parameter space of the instruments is often huge and rarely explored. Put simply, there is a vast semantic gap between the composer’s conception and the mechanisms for realization. In this paper we explore a way in which modern computer science — specifically eScience — may be able help.

2. USING E-SCIENCE TECHNOLOGY IN MUSIC

At first sight there seems to be little in common between eScience and the creation of music. However they are both capable of generating large datasets and utilizing significant computational resources to create those datasets. Furthermore, in the same way that scientists collaborate on experiments and share scarce resources, it is common in the electronic music community to share software and more specifically new synthesis components. It is now becoming the practice to describe and publish a facility for (semi-)automatic discovery by software tools that support eScience workflows (chains of services feeding from one to another each carrying out some transformation on the data flowing through the network). Likewise, it would seem desirable to be able to do the same with a synthetic instrument, rather than uploading it to a repository, letting others download it and then having to spend time helping with configuration on an unfamiliar platform. Not only is it often impractical to install all components on client machines because of size, but also maintenance becomes a significant issue. These observations and parallels make it appropriate to ask the question whether the same technologies, namely (semantic) web services and (semantic) grid technology, that are enabling in silico experimentation can also be brought bear on synthetic instruments, to try and narrow the (semantic) gap between computing facilities and user objectives, and what benefits might result.

3. WEB-SERVICES AND WORKFLOW ENACTMENT

Web-services[5] can be described as self-contained software components that provide a well-defined function. Deployed by tools such as Apache, Tomcat and Axis, they allow authors to publish software but maintain control. The important point here is that the component is shareable but not distributed from the host site. Users typically access the web service using a messaging interface such as SOAP over HTTP[9], and so they do not need knowledge of how the service is implemented, merely what parameters are required and what kind of result is returned: it is a black box. This organization effectively decouples the client and the server both temporally and spatially. It also serves to remove client/server platform dependencies.

At the primitive level, the client might write a program that opens a connection to a web service, sends some data and waits for a result, rather like a remote proce-
duction call, only even slower. But the reality is that, at least in eScience, users wanted to be able to join together collections of web services into workflows — a process sometimes called orchestration. Thus it became obvious that the plumbing or patch-bay metaphor used in image-processing toolkits and some sound synthesis toolkits (e.g. Kyma [16]), since the 1970s offered a familiar and potentially effective solution for the graphical programming of web services. In fact, the workflow design is decoupled from its enactment (execution), the latter being described in one of several languages such as SCUFL (used in Taverna) [17], BPEL [3] and OWL-S [14]. These enactment languages allow many atomic services to be combined together into complex applications using common workflow constructs (e.g. loops, conditional branching) to control data and execution flow. Furthermore entire workflows may themselves be deployed as single components in other workflows.

4. OVERVIEW OF A SERVICE-ORIENTED COMPOSITION ENVIRONMENT

Building on ideas and tools from e-Science we have developed an initial service-oriented composition environment for music. There are three components to this environment. We have based this system on the Csound synthesis language[2] and so have inherited the MusicV distinction between orchestra and score:

1. The first component is a collection of synthesis web services, that incorporate a selection of the basic atomic components required for sound creation and processing, such as oscillators, wave-table generators, physical models and vocoders, constructed from Csound (see section 5).

2. To this is added a tool that gives a description of music in terms of the construction of the instruments (parameters, connections) and the score they will play. This is very similar to normal signal flow editors like Patchwork[15], but the underlying implementation is very different.

3. The final component is the environment of use in which we can build and invoke instruments from the individual services. At its most basic this could just be a client program that invokes one or several web services, but the motivation behind doing this work was to avoid that level by using workflow editing and enactment tools. Because of the standardized nature of the service interfaces, any tool will do the job; we experimented with Triana[11] and Taverna[13]. Space precludes a detailed comparative assessment here, but both offer their share of good features and problems.

The current implementation is a prototype “proof-of-concept” but we can already see a number benefits that we anticipated: specifically, the user requires no local software installation beyond a standard web browser and the workflow tool, and client processing power is no longer an issue for the creation (off-line) of audio files. The system currently only supports web services built from Csound, but as far as we are aware, there are no platform dependencies in the architecture to prevent the integration of services that use other systems like Max/MSP[1] and SuperCollider[12]. More work too is required to address ease-of-use when constructing complex workflows — as Figure 2 shows, even quite simple tasks demand large numbers of connections and intermediate services.

Having outlined the general idea, we now turn to the detail of the basic building block of our system, namely how to deploy a specialized Csound as a web service.

5. BUILDING CSOUND WEB SERVICES

To demonstrate the validity of the approach for a large range of Csound opcodes, we constructed a tool that automatically generates the web service interface code based on the opcode specification. This is not because we expect users to invoke a web service with all its overheads for some of the trivial operations, but a common pattern of use in Csound is to design an instrument such that its interface is just an opcode. It is also certainly the case that Csound users will not want to write Java interfaces to web services (or debug them). Thus, we have created individual Csound opcode services for each opcode, with its name and interface and function matching the underlying opcode. Csound score files are also used. There are over 1,200 opcodes in the current Csound 5 not counting user-created opcodes. Manually writing (and maintaining) interfaces for all those opcodes is clearly not practical, hence the motivation for constructing a tool. The tool is quite complex because of both the large number of parameters the some opcodes accepts and the ways in which they can be used creates conflicts with the type signature specification of the web service. It is also incomplete in that it can only handle a (large) subset of the opcodes: the remainder may readily be supported either by extension or by direct custom implementation.

Each opcode service performs four main tasks: Decode the input message, Generate Csound files, Invoke Csound and Return the Csound output.
Csound’s flexible interconnection model presented a challenge for the implementation. We encode all types of data (audio or control data arrays, numeric constants, references to score fields and tables) within binary arrays which form the inputs and outputs of the service. The service inspects incoming data to interpret the input types it contains. This is transparent to the user.

Then depending on the types of input found the service has to generate valid Csound orchestra and score files, which includes creating calls to pass data between the service and a local Csound using the Csound software bus.

The actual audio processing is done in Csound, which is invoked from the Java service using the Java implementation of the Csound5 API. This controls the compilation and performance of the generated orchestra and score files, and collects the input data to Csound as required and capturing the output, again via the software bus.

The final result, whether audio or control signals is send to the requester, encoded as binary data.

**Implementation details:** We briefly describe the implementation of the service which invokes the Csound opcode oscil. Whilst some aspects of each opcode service implementation obviously differ, in general the differences are minor. oscil is a signal generator modelling a simple oscillator, taking three input parameters:

- **kamp** The amplitude of the output sound (k-rate)
- **kfreq** The frequency of the output sound (k-rate)
- **ifn** The number of the Csound function table to read from (i-rate)

oscil has one audio output (i.e. at a-rate), the naming of which is unimportant, but the name ares is often used in Csound examples. Taking into account the additional required information required above, we see that the complete interface for the oscil service implementation in Java is:

```
byte[] oscil2 (byte[] kamp, 
byte[] kfreq, 
String ifn, 
String number,  
String orchheader, 
String score)
```

For other opcodes the interface will differ according to the the underlying Csound opcode’s interface. There is no limit to the number of input parameters, but a restriction in the current implementation is that only opcodes with a single return value can be supported.

**Deployment and Use**

Once the services are deployed on a server (e.g. Apache Tomcat / Axis) they can be consumed by any client — programmatic or visual. In searching for a suitable client to demonstrate the services, we found that GridPSEs offer a number of benefits.

This is a familiar visual environment for composing synthesis units with extensive workflow capabilities. This provides support for many Csound language control constructs that are not included in the opcode service implementation. As we desire, workflows (i.e. instruments and orchestrations) defined in the GridPSE can be collected together and deployed as single composite services.

Figure 1 shows a simple synthesiser modelled in the Triana system, and figure 2 is a similar process in Taverna.

### 6. LIMITATIONS AND DRAWBACKS

There are inherent problems with web services. Inter-service communication is expensive, not only because of bandwidth, but also computationally. This because encoding and decoding take a significant load in the middleware layer. In addition the actual data flow may be less efficient than the workflow suggests, especially if a a central dispatcher and controller is used. The other major problem is that the ‘natural’ interpretation of these processes is as a stream of data, or asynchronous invocation. While this is possible in web services the supporting tools are complex and as yet immature. At some stage a streamed version will be necessary to make web services really useful in sound synthesis. It may be possible to create a pseudo-stream by dividing the data into short packages, in effect exploiting the control-rate structure of MusicV, but at present that is not operational. There is also a limitation in the use of global data; data passed to the services needs to be explicit so it can be controlled by the Java layer.

### 7. RELATED WORK

The work incorporates the ideas of Netsound[4] of remote synthesis\(^1\) and of NetCsound[6] as a web-based service, but goes significantly further in distribution and identification of resources. Netsound allowed remote synthesis and performance, and these idea have been largely incorporated in current Csound. NetCsound is closer in concept as it allows synthesis elsewhere to the local service, without the installation of special software. The scheme here adds distribution to this format, avoiding computational bottlenecks, and incorporating extensibility.

\(^1\) Including SAOL http://sound.media.mit.edu/mpeg4/
The use of signal flow diagrams is of course not new, well used in Kyma and Pure Data for example. Our user interface is thus built in well known and familiar mechanisms. The detailed system is close to Pinkston’s Patchwork, but the compilation process does not generate Csound orchestras but rather orchestrates the web services.

8. POTENTIAL APPLICATIONS & FUTURE DEVELOPMENTS

This is just the beginning of the possible use of this technology. We envisage the composition of services leading to an increased composer efficiency, allowing easier reuse of instruments and sounds, via the availability of workflows.

An intriguing possibility is to use grid synthesis, effectively distributing the computation of musical synthesis, and this making computing power, at least for off-line works, not an issue.

In the future we view the introduction of streaming web services as very important. There are signs that the general web-services community are working towards this, for example with Axis2[8].

The other component we have not yet developed is the resource description and discovery subsystem. With an agreed interface there is no reason that the individual web services should be Csound based. The introduction of brokerage services will give a mix’n’match system. Currently we expect to develop the Open Middleware Infrastructure Institute’s Knoogle broker[10] using both technical and folksonomic descriptions of services.

The other underdeveloped component is the use if agents. We envisage that in the near future agents will be able to explore the large parameter space of an instrument (or workflow), and under human guidance to inform the use of these synthetic instruments[7]. This is an extension the synthetic performer concept[18]. Developed from that there is the possibility of a performance ontology for agent communication, and hence reducing again some of the fine-detail crafting of music. Agents may even be able to form and manage ensembles with minimal human intervention.

9. REFERENCES