User Participation in Software Configuration and Integration of OpenMusic, Humdrum and Rubato

Jörg Garbers

Abstract

The field of computer aided music analysis is highly interdisciplinary. Though the purposes of applying the computer are quite diverse, independently developed software tools can often be shared, and their use can open new interesting research fields. In this paper I give an overview on design issues within the software packages *OpenMusic*, *Humdrum* and *Rubato*®, that allow for user participation and configuration to cope with the diversity of user interests. I show by examples that the integrated use of these packages is reasonable but can be technically challenging. I claim that software integration components, which we build into OpenMusic, can significantly simplify these integration tasks.

1 Introduction

People working on computer aided music analysis come from different disciplines like musicology, semiotics, mathematics or computer science. Not only researchers from different disciplines typically have different points of view and different foci, but even within a discipline the purposes of applying software are highly diverse. They reach from retrieving-and-displaying-information over proof-of-concept-by-implementing to interactive-theory-shaping, and for some researchers music is also just an interesting complex field for testing general methods and processes like statistical or learning-based methods.

1.1 Roles in software development and configuration for music research

Classical software development models, methodologies and principles are often made for large projects, where there is a customer, who orders a software product, a group of software developers, who build the system and a possibly large group of users, who will use the software for production purposes or services.

In typical software development for (academic) music research however, there are no paying customers, just a small number of users and very few professional developers to do the implementation for the researcher. The motiva-

tion for building the software derives often directly from the subject (music) which is modeled in the software, and often the developer is also its most important user. This is also the case in typical OpenSource software development. In combination of the functions the researcher/developer seems to be most productive, as he or she may be interested both in the musical data, in the processes (shaping-by-implementing) and in the implementation (performance, maintenance, etc.). As a consequence software development methodology is typically secondary and the development process results in working prototypes or collections of small tools rather than easy to use integrated products.

The software packages Humdrum, Rubato and OpenMusic, which are examined in this paper, show a common property to cope with this situation and to be useful for as many users as possible: They derive their main value in the hands of the user, by showing different kinds of flexibility, which enable the user of the software to develop new processes and thereby perform genuine new research without the need to consult a professional developer.

1.2 Kinds of configuration

Flexibility of tools can be achieved by deploying the following kinds of configuration techniques:

Preferences: In this most common kind of configuration the user specifies values that in the following affect the behaviour of the software. Changing the "preferences" of a program is a typical configuration activity, but also the actual setting of values that affect a computation can be seen as a configuration act. This configuration kind is heavily used in Rubato.

Selection: This kind of configuration deals with the mapping of functionality to tools and the restriction of functionality. A simple example is the mapping of file name suffixes to programs in the operating system, so when a user double clicks a file, the corresponding program is started and the file is opened as a document of that program. Another example of this type of configuration is the JRing system, where the musicologist configures the system by defining, *which* functionalities he or she wants to use in the next JRing session. The JRing system on the other hand *maps* functionalities, such as search functions to available tools (Humdrum tools or other plug-ins). (See Kornstädt (2002).)

Combination: This kind of configuration is functionality-based. The user is offered handles (components) to the functionality of the system and can flexibly combine and control the data flow between them. Well known visual programming based configuration systems are MAX and OpenMusic. Similarly interpreters give access to (exported) functionality of the application through programming in a scripting language. This kind of configuration is available in each of the examined software packages.

1.3 Integration preconditions

The combination and partially also the selection type of configuration are based on several assumptions and preconditions about the components which are to be integrated:

Accessibility: Each component should be fully accessible, because if it is only usable in a specific context, for example as a subfunctionality of a bigger component, or within a fixed predefined configuration, it might not be possible to use its full parameterization possibilities. To gain full accessibility, each component must be either "exported" (made available) explicitely, or a scripting component generically exporting any existing component living in its environment must be deployed.[1]

Common environment: In order to combine two components technically, they must share a common environment. This can be the address space of a program, a file system or one or more operating systems of a network. Components can be combined programmatically or manually by performing actions. Note that the performance of passing data and control between the components within the environment depends on the tightness of their integration. It can be an important factor for the usefulness of the integration: Consider an integration by hand, where the musicologist for each of 500 scores must first start an application like Rubato, adjust the settings, produce a result, save it, then start another application, do something there and save the result. This way, if not sufficiently automated, corpus-based analysis can be very tedious and error prone.

Data coherence: A precondition for combining two components is that the type of data the calling component (or the calling environment) provides, matches the data, that the called component expects. Mathematically speaking: The concatenation of two functions is only possible, if the codomain of the first identifies to or is a subset of the domain of the second. In computer science the situation is more complicated than in mathematics, because domain and codomain include not only the semantical type (e.g. a number), but also the representation type (e.g. an integer or a string), as well as the environment type (see above).

2 Humdrum (basic score processing)

The *Humdrum toolkit* is a collection of text based command line tools for corpus-based music analysis. The Humdrum data format and the tools were designed with ease of use and extensibility in mind. Their purpose is not more, but also not less than to serve some basic needs of musicologists interested in corpus-based music analysis. In contrary to Rubato, where a tool represents an innovative theory which is still to be validated, the tools here typically are much smaller and represent well known relations[2] and are only made "to do things quicker". Nevertheless complex and interesting tasks can be performed by choosing adequate data representations and combining existing tools. Therefore the user must know his or her tools, their applicability and limitations.

1 Note that full accessibility might violate some software development principles, such as encapsulation (information hiding). Commercially available software programs typically only export a limited set of it functionality for several reasons. When a programming interface changes in a new version, user defined scripts could easily become invalid and cause frustration. But if users are too content, because they may extend the software by themselves, they would probably also not buy a new version. As we will see, the examined non-commercial software packages follow an approach of higher accessibility.

2 E.g. a converter from a certain notation format to another format.

Anyway, one important part of the Humdrum philosophy is that the user is responsible for checking the results.

The Humdrum tools use the file system and UNIX-shell typical techniques like *pipes* and *environment variables* for data in- and output. Combined this way in *shell scripts*, the techniques can be seen as realizations of hierarchically structured data flow models: a shell script can be – like any basic command – a node within any shell script. Empowered by shell script programming the Humdrum commands follow the UNIX paradigm that the tools should be kept simple but easy to combine.

Humdrum files have a common row-and-column based syntax, which is easily accessible by a lot of common UNIX tools like *grep*, *sed*, *wc*, etc. and by programming languages in general, especially in text processing languages like *AWK* and *Perl*. The semantic of a token (text field) within Humdrum files is determined by syntactic relations to other tokens, especially by one or more interpretation tokens preceding it in the column (spine interpretation tokens). Besides of using well supported token formats[3] for music notation, such as `**kern` or `**MIDI`, new token representations can be invented by the user to represent additional aspects of music and/or to provide data for newly developed tools. (See Huron (2002).)

2.1 User and programming levels

The Humdrum tools can only display their full usefulness, when combined and expanded in creative ways. So user participation in software configuration (combining two tools) and development (developing a new tool) is highly recommended in the Humdrum world. In fact, because there is no Humdrum frame application and no need for GUI components (like in Rubato or OpenMusic), any working process or programming task can allready be performed by suitably skilled musicologists and not just by professional software developers.

The most important productivity factor in using Humdrum is the ability of the user both to map knowledge about musicological working processes to tools and representations and to become inspired by the technically possible. (Often the knowledge of techniques leads to new implementable ideas.) Becoming more experienced with the Humdrum tools, users typically advance and show increasing technical skills of usage and programming:

- Beginner: uses only a few tools and command line parameters, when running the shell interactively uses redirection to files and removes of temporary files manually (see below). He or she builds simple shell scripts.
- Advanced user: uses many tools (both Humdrum and UNIX), utilizes the flexibility of the tools (command line parameters, environment), bases the interactive use on pipes and automatic clean-up of temporary files. He or she builds advanced shell scripts and small AWK scripts.
- Designer/Developer: designs new representations, and new commands in a clever way using appropriate programming languages (AWK, perl, c, java, etc.), regards existing efforts (maintainability).

3 Well supported means: there are tools that make use of the semantic meaning.

As an example consider a beginner's and advanced user's way of comparing harmonic and melodic intervals:

Beginner's way of using Humdrum:

```
hint chor024.krn >temp.1
mint chor024.krn >temp.2
assemble temp.1 temp.2 >out
rm temp.1 temp.2
```

Advanced user's way of using Humdrum:

```
assemble <(hint chor024.krn) <(mint chor024.krn) >out
```

2.2 A distributed support model

It must be stressed that – not only because of the comparably difficult usage of a UNIX shell, but also because of the high potential of the tools – to use the tools effectively, users have to spend some time in learning the tools and skills. Luckily there are already useful references:

- As a beginner one should at least follow the tutorials and try and modify the given examples. This will lead to a sense for the tools.
- As an advanced user one should read manual pages and foreign code to quickly develop own solutions.
- As a designer/developer one should study the architecture of Humdrum commands and representations in order to design representations and write good tools that can be of use for others, and one should keep maintainability in mind.

Currently the Humdrum tools are collected by David Huron and Craig Stuart Sapp, thoroughly testet and released as a simply installable package. While this effort must be highly appreciated, both cannot be expected to give general support for the Humdrum tools. Therefore, from my point of view, central communicational infrastructures such as mailing lists should be established, where users can support one another. Advanced users may also argue that development and maintenance of their own new tools should happen in collaborative development environments, such as professionally hosted OpenSource-projects (SourceForge etc.).

3 Rubato (interactive theory testing)

Rubato is an interactive graphical software tool for music analysis and artificial performance (see http://www.rubato.org). It was developed by Guerino Mazzola and Oliver Zahorka on the NeXTStep Operating System and ported by us to OpenStep and Mac OS X, where we develop it further in an OpenSource project. [4] A main idea of its design was, that it should be easy to use

4 The reader should not be confused by the fact that there are two complementing software development activities operating under the label "Rubato": This paper and the three articles by Noll, Fleischer and Nestke in this volume refer to a collection of classical windows-and-buttons based interfaces and Objective-C modules for music analysis and performance available for Mac OS X, whereas the paper by Göller and Milmeister refers to a collection of Java modules for data visualization and manipulation, for software distribution and operation and for music analysis and performance, not yet publicly available at the time of this writing.

by providing a windows-based graphical user interface, people are used to, and that it must be extendible. Following a component based approach, it accepts modules created by other research and development groups to allow a discourse which is competitive in theory and collaborative in terms of tool usage.

Rubato's architecture is that of an application framework with a frame application and dynamically loadable plug-in modules, so-called Rubettes®. The frame provides basic data storage and data access capabilities and defines and establishes protocols for loading Rubettes, setting up their graphical interfaces, and accessing their data and functionalities. A Rubette by contrast provides a graphical interface to its functionality, giving the user the opportunity to change some theory settings, calculate results for a given musical score and examine them interactively.

The implementation language and development environments for Rubato and the Rubettes are *Objective-C*, the Macintosh *Cocoa-Frameworks*, *Interface-Builder* and *ProjectBuilder*, important tools for building applications on the Mac OS X platform. The embedding of FScript[5], a scripting component for accessing the Objective-C runtime environment, opens Rubato and its Rubettes for many kinds of customizations.

3.1 Configuration levels

Configuration in Rubato is more difficult than in Humdrum. The reason for this is that – differently from a Humdrum tool – it has a state, on which the results of a scripting command depend and which in its own may be changed by scripting commands. Compared to Humdrum's shell-scripting, on the one hand scripting in FScript could work in a much finer granularity[6] on the other hand Rubettes as reasonable scripting targets are much more complex and difficult to program.

The following customization and programming levels can be identified:

- Visual configuration: That Rubettes have a state can be exploited by the frame application in so far as newly loaded Rubettes can be put in a previously saved state. This way a collection of configured Rubettes could be loaded and applied to new musical material.[7]
- Script additions: Rubettes offer a programming interface for setting up some values and user defined hook functions that affect the behaviour of the Rubette. When calculation takes place, the Rubette looks up the value or executes the function and uses its return value. The author of the hook function must only know the programming interface and of course the scripting language FScript. This type of configuration method is heavily used in the current version of the HarmoRubette,[8] which allows to specify so called harmonic spaces and custom calculation methods. Even the graphical user interface is affected.
- Script automation: Because scripts can be used to simulate user actions, or to invoke any method of any programming class, nearly any func-

5 http://www.fscript.org
6 Any method of any runtime object can possibly be called.
7 This feature is not implemented yet.
8 See paper "New Perspectives of the HarmoRubette" by Noll, Brand and Garbers in this book.

tionality of Rubato can be accessed by the scripting environment. This possibility can be exploited for semi-automatic software testing, but also to automate a process, where the user would have to iterativly enter values and calculate (and save) some results (e.g. graphics). It was successfully demonstrated with the MetroRubette to show how "the grass grows" in the WeightView window, while the minimal length of local meters decreases. Necessary for a successful application of this method is a knowledge of the (public and private) programming interface of a Rubette.

- Patches: A method for applying bug-fixes or for adding custom properties of the frame application or a Rubette, while staying synchronous with the main development process. When starting or when loading a Rubette, Rubato looks for such patches in the file system. To produce a binary patch, a developer compiles the difference-code[9] into a binary bundle. The Objective-C runtime allows the patch to extent classes or overwrite methods of the Rubette.

Besides these customization possibilities, a developer can write a new Rubette from scratch or based on the source code of an existing OpenSource-Rubette. The availability of the source code furtheron allows to adopt the frame application or to add new Rubette communication or persistence techniques, e.g. for data interchange with databases.

3.2 User and developer resources of support

Similar to Humdrum, there is only limited support available for source code related questions. We also expect more interest in the FScript configuration possibilities, when they are shown to Rubato users, who get in touch with the limitations a graphical user interface imposes on the user actions (script automation), or to those who want to explore own calculation methods in an existing user interface (script additions).

Sources of reference for users and developers are:

- Example scripts made by other users or developers for similar tasks.
- The Rubato programming interface descriptions.
- The Rubato source code and user interface files (NIBs).

It should be noted that feedback from the users concerning FScript desires can be a valuable resource for deciding, which API (e.g. for hook functions) and what documentation is needed in future versions of a Rubette.

4 OpenMusic (composing)

OpenMusic is an interactive software tool for composing, developed by Carlos Augusto Agon and Gérard Assayag (see Assayag et al. (1999)). The main idea for its creation was that composers who want to use the computer for creating (generating) musical material can in principle not be satisfied by a fixed tool. Also a plug-in architecture like Rubato does not satisfy a composer since it is too difficult to combine the components in unforeseeably creative ways or

9 Not written in FScript but in Objective-C.

to create new components. Only a programming language is flexible enough to satisfy the growing modelling demands of a composer using software in a creative way.

OpenMusic consists of a visual programming environment (the core) and a collection of packages for different musical and non-musical programming and visualization tasks. The core defines the visual programming language, a framework for the main user interactions on graphical objects and means for storing and loading user defined constructs.

OpenMusic is written in LISP and currently needs the Macintosh Common Lisp (MCL) environment. Because OpenMusic transforms visual programming constructs into (textual) LISP constructs, it is able to offer not only OpenMusic dedicated programming elements (music functions, control structures, score editors) to the user, but also to map existing elements of the base language (functions and classes written in LISP/CLOS[10]) into the visual environment. A part of the MCL programming environment is include in OpenMusic, providing a help system for LISP constructs, a LISP source code editor, etc.

4.1 User levels and roles in configuration and programming

The role of a composer in OpenMusic is that of a programmer. At first the activities within a patch[11] can be seen as configuration activities. But when making an abstraction from a patch in order to use it as a building block in another context these activities turn seamlessly into programming.

The following levels of usage and programming can be identified:

- Beginner: uses a few components within a patch and linear programming structures.
- Intermediate user: uses many OpenMusic specific and common LISP components, uses recursive programming structures, understands the control and data flow and some of OpenMusic's limitations.
- Advanced user: uses external LISP packages and "more organized" modeling techniques like generic functions, classes and variables.
- LISP developer: defines functions, methods, classes, variables, macros and packages for LISP contexts.
- OpenMusic visual box developer: defines new user packages with visual OpenMusic components like patch boxes and factory editors.
- OpenMusic kernel developer: defines OpenMusic's meta object protocol, kernel and user interface behaviour, works on the user language design. (See Agon and Assayag (2002).)

The distribution of programming activities among developers and users is displayed in figure 1.

10 CLOS: Common Lisp Object System.

11 Patch: A working environment in OpenMusic. Note, that this usage of the term patch is not closely related to bug fixing patches e.g. in Rubato.

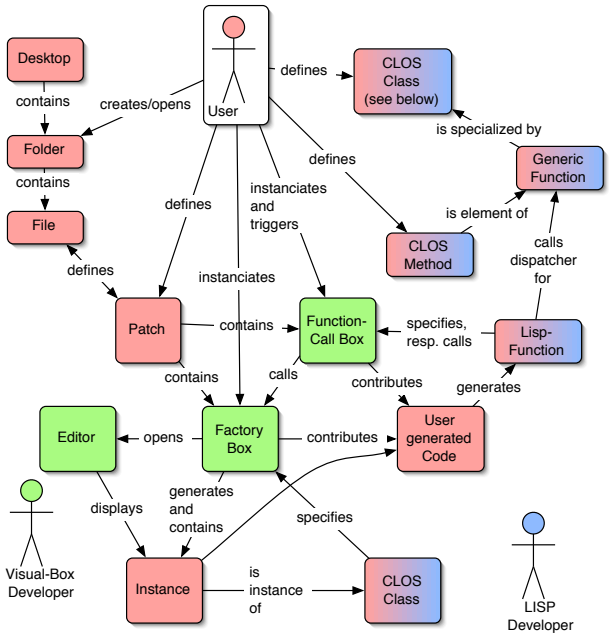

Figure 1: Programming and usage activities in OpenMusic

4.2 Support at IRCAM

Compared to Humdrum and Rubato, there is a comparatively large and well-organized user community. Within the IRCAM, composers have the opportunity to work collaboratively with OpenMusic developers and other advanced users. IRCAM also organizes workshops twice a year, where external users can learn more about new features of OpenMusic and other music software. Users have also the opportunity to present their OpenMusic related work, such as compositions or generally useful developments. This way users get in touch with each other and exchange knowledge and opinions about the usefulness of tools. There is also a mailing list.

Other sources of reference are:

- The tutorial and the user guide for beginners.
- The reference guide and the component related documentation provided by the help system for functions, classes, etc. for intermediate and advanced users.
- The LISP source code for LISP developers.
- The OpenMusic developers guide and the complete OpenMusic kernel source code (with comments) for OpenMusic developers.
- An architecture overview for kernel developers.

5 OHR (software integration)

What should have become clear so far is, that each of the software packages Humdrum, Rubato and OpenMusic provides a very flexible way of configuring by programming. Nevertheless teaching experience shows that some packages are easier to program for the users than others.

Besides of this pragmatic aspect, each software package provides some functionality that is missing in the other software packages. In the following I give an example of the integration of processes including elements of all packages, that have not been done before.

The direct solution requires much knowledge about programming in each of the software packages. So in the remaining sections I show different integration techniques, which I built into OpenMusic in order to generate a highly configurable software integration platform, which can be used to combine functionalities from OpenMusic, Humdrum, and Rubato more easily.

5.1 An OpenMusic, Humdrum, and Rubato integration example

Task: For each Bach chorale from the CCARH[12] collection of Bach chorales, an automatic harmonic function-theoretic analysis must be made for each first phrase. The results must be compared to manual annotations. Differences must be presented.

"Manual" solution: Knowing the functionalities of Humdrum, OpenMusic, and Rubato, one can come up with the following process:

- The scores can be obtained in Humdrum's Kern format from the CCARH repository.
- Humdrum commands are used to extract the relevant parts from the scores.
- A converter is used to transform each result into Rubato's EHD-Notation. The outputs are stored as files.
- The HarmoRubette is successively fed with each of the files by use of a script. The theory preference settings for the Rubette are made with a different script, stored also as a file. For each input file, the main script is used to output a file with the "best" harmonic path with respect to the theory preference settings.
- The annotation file (e.g. in Humdrum `**harm` format) and the Rubette output file are read and compared. The comparison can be made within any of the software packages.
- The differences could be visualized in the Harmonic Path view of the HarmoRubette by scripting the graphical interface, in OpenMusic in common music notation as highlighted chords within a Chord-Sequence editor, or in Humdrum by producing another `**harm` spine and assembling the annotated and obtained spines.

12 Center for Computer Assisted Research in the Humanities, Stanford University, California.

"Automated" solution: The whole process can be automated by creating a global script in any of the programming languages, which would forward different tasks to the different software packages. This can be done

- by (remotely) accessing the script interpreter of each one of the software packages (the shell, Rubato's FScript component, or MCL's eval-server)
- via some kind of interprocess communication (system calls or remote shell invocation, distributed objects calls or Apple events)
- by feeding the respective interpreter with function definitions and by triggering (evaluating) these or the builtin functions.

Each scripting environment allows to extent the runtime environment of the host application by defining functions or commands. In state preserving situations (OpenMusic and Rubato), such functions can be simply given a label, which is used to reference the function in further function call requests. In a situation, where a new interpreter is started each time a different software package calls (see section 5.2), the function must be installed in the host system, which in the case of UNIX is the file system, where a file can represent an executable command. This is accessible by the following instances of a shell interpreter by use of its path in the file system.

Problems: The proposed solution requires not only the knowledge of the components available in each package and their usage (parameterization), but also the scripting languages of the software packages as well as the interprocess communication issues. While the first requirement is equivalent to knowing *what* can be done semantically and thereby defining a condition for effective work (see also section 2.1), methods should be elaborated in order to shield beginners from having to know the different *how-tos* at a higher programming level, while more advanced users still can intervene at a lower level, to provide better solutions.

5.2 Interprocess communication routing

To establish a communication between two programs, a route must be found from the caller to the called program. It can be realized as a chain of suitably configured communication components. This can be very simple if the two programs speak the same communication language, such as *Apple events*. As a more complicated example, we consider the communication between OpenMusic, running on one computer under Mac OS 9, and a Rubato process, running on a Mac OS X computer. As remote Apple events may be disabled on the Mac OS X, or else the user does not know how to send events from OpenMusic to another computer, an indirect route must be chosen.

In MCL the OpenTransport facility of Mac OS 9 can be used. This allows to open a TCP/IP connection to another computer. In the MCL community a user has implemented the remote-shell protocol, such that a command can be sent as as string to the Mac OS X computer, be executed there and return as a string it's standard output. Arrived on the other computer with correct user access rights (the user name and password are sent at the beginning of the protocol), the command can open a local communication channel to Rubato using Apple events or Objective-C's distributed object calls.

To provide different ways of communication, we implemented some file-based and/or event-based interprocess communication processes for different situations. For detains see the OHR[13]-project[14]. As now different communication elements are available, the task of creating a working chain of communication channels from one program to the other can again be seen as a configuration problem, which could be modeled in any of the scripting languages provided by OpenMusic, Humdrum or Rubato. This could be done by anybody who is aware of the actual situation and the routing elements.

5.3 Language mappings

One way (proposed in Garbers (2003)) to reduce the need to know different programming languages, or at least their syntax, is to extend OpenMusic's processing methods for patches, which are visual configurations of programming language elements. Currently these are mapped to LISP expressions (function definitions, function calls etc.), but they could be mapped to other programming languages and runtime environments as well.

This would release the users from having to know the concrete syntax and processing details of different programming languages, which can already be tedious for different shell scripting languages like *csh* and *bash*. E.g. in *bash* there are two for-loops, with syntax

```
for name [ in word ] ; do list ; done
```

and

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

while in *csh* only the form

```
foreach name (wordlist); ... ; end
```

is used. Besides, the correct construction of more complex shell scripts by hand is allways challenging in any shell language, because of the diverse substitutions for shell variables, regular expressions and escape sequences that are processed by different entities (see the manual pages for details).

A language mapping would also allow to provide the user with higher level language constructs, which are mapped by the system to possibly lower level language constructs available in concrete scripting languages. E.g., the LISP-typical *mapcar* function for applying a function to each of the elements of a list could be mapped to a *while*-loop in a shell script, that reads lines (the elements of the list) from a file and feeds them successively to a command, which realizes the function to be mapped. The command itself can be elementary in the target software package, or could be derived from an OpenMusic user patch.

5.4 Configuring an interpreter mapping in OpenMusic

More precisely, this is realized on an easy to use and flexible to configure language expression mapping system that we have embedded into OpenMusic. The user – as usual – connects elements within a patch, specifying the "what" of a process. Besides this, he or she specifies within the patch one or more

13 OHR: OpenMusic, Humdrum, Rubato.
14 http://flp.cs.tu-berlin.de/MaMuTh/Ohr/

"interpreters" by name, which realize the meaning of the associated visual elements, either – when evaluating an element within the patch – as an actual call or – when an abstraction of the patch (a function) is made – as code that becomes a part of a script.

An "interpreter" must be configured by specifying

- a service provider, e.g. a remote application like the FScript component within Rubato, that provides the required functionality,
- the target programming language, usually the scripting language, the service provider understands,
- initialization routines and wrapper code needed to evaluate the scripts in the right environment, e.g. within a namespace reserved for use from OpenMusic or from a specific OpenMusic patch,
- a transportation service, which brings the code and the calls from OpenMusic to the service provider, e.g. via TCP/IP and remote shell commands, or via Apple events, (see section 5.2)
- a data converter to map input and output parameters that are passed *by value* between OpenMusic and the service provider,
- a parser that knows how to interprete the result coming from the service provider (e.g. split lines and return elements as a list of strings or numbers, or evaluate the returned text as a LISP expression),
- a language construct converter to map higher level programming structures to those understood by the service provider's scripting environment,
- a language element converter to map between visual elements and available elements in the target language/system,
- a script generator, that is aware of the different syntax duties (which sign finishes an expression, which line feeds are used, how to handle escape sequences etc.). It can optionally produce more human-readable code by doing some pretty printing. Scripts produced this way are more maintainable and can be of use in the target application even without OpenMusic. They can be given to users who do not have OpenMusic but want to use and adopt the produced scripts.

It must be noted that typical computer science skills are needed to realize these basic functionalities. However, the concrete configuration can be performed by advanced users, and some parametrizations, like entering the host name, user name and password of an account on a remote machine, even by novices. A lot of configurations can be made by users, if there are similar example configurations.

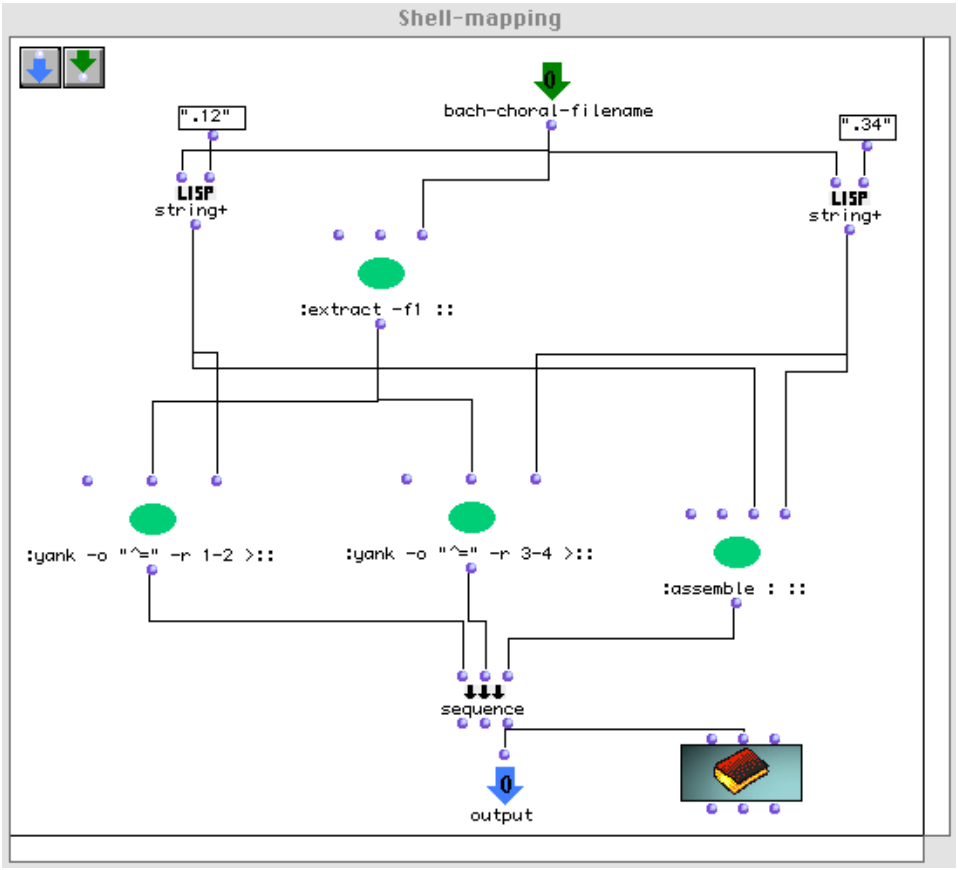

Figure 2: Humdrum commands in an OpenMusic patch

5.5 Examples of mixing programming languages in OpenMusic

Figure 2 shows an example of an OpenMusic patch that is directly mappable to the following shell script:

```
#!/bin/bash
extract -f1 $1 | yank -o "^=" -r 1-2 >$1.12
extract -f1 $1 | yank -o "^=" -r 3-4 >$1.34
assemble $1.12 $1.34
```

The script extracts the first voice of a Kern score file, the name of which is given in the first argument. On the result it applies an operation that selects a range of barlines. Two such selections are pasted side by side for whatever study purposes. The data passing techniques used here are file redirection (through `>`), file referencing and the use of pipes (with `|`). The control flow techniques are linear (pipes, no keyword necessary in the patch) and parallel (`sequence`

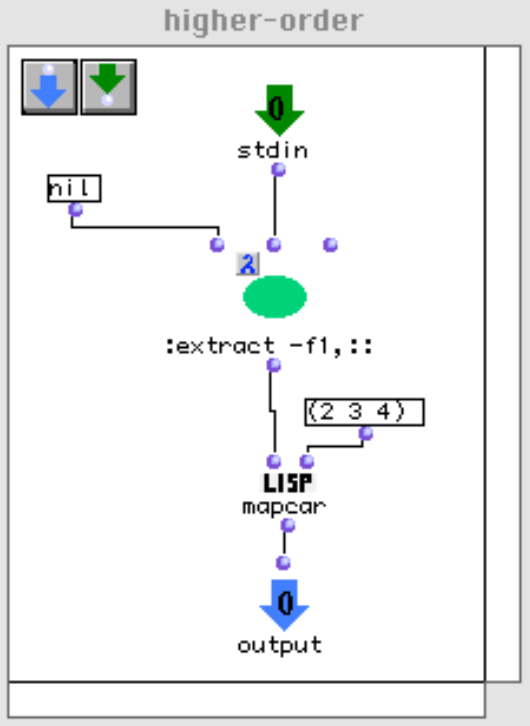

Figure 3: Higher order LISP constructs with a foreign Humdrum function

keyword in the patch, newlines in the script). Note that there need to be only one box for the two `extract` function calls.

In this case, the string concatenation function `string+`, which is known to LISP function users in OpenMusic, is simply mapped to a shortcut in shell scripts (just joining the arguments). This trick allows us to use the patch in two ways:

1. In test modus, the data flow is controlled by OpenMusic, which directs the data along the connections from one box to the other. By evaluating different boxes, the user gets a sense for the parameters of the commands and adjusts them until he or she is satisfied. Very handy are text boxes (the box with the book icon), where the standard output of a command can be captured and inspected.
2. In application modus (i.e. when the patch is called with an actual parameter), the patch is compiled into the shell script. Since the patch in this case is totally mapped to a shell script, it can run completely in the Humdrum environment, without the need to control the pieces from OpenMusic any more. So nested Humdrum patches can run very efficiently, making this technique especially appropriate for remote processing.

Figure 3 shows another example, where a LISP-skilled user applies a Humdrum command in a LISP style. From a given input text in Humdrum format

the patch extracts two columns (e.g. two voices). It builds and returns a list of the outputs for column numbers (1,2), (1,3) and (1,4). The user applies more advanced language features of OpenMusic, namely the lambdafication of a function, where input slots that have a connection are bound at lamdafication time and the remaining slot is bound at calling time. The mapping of such a construct to a shell script is not trivial, it would also comprise decisions about how to represent a list of texts in the target environment (the shell script or the file system) and how to pass it back to OpenMusic, if the patch is called from another patch.

5.6 Wrapping up functionality

A good user experience is often coupled with an intuitive packaging of functionality. This is especially true when, as in the integrated system of OpenMusic, Humdrum and Rubato, there is a lot of functionality. Since the underlying usage paradigms are quite different, adequate programming patterns will be needed for their access (though the programming language is the same visual language).

Luckily, in OpenMusic it is easy to wrap functionality by defining a function with the help of patches (nameless) or generic functions (nameable). Users not interested in their details can use them as "black boxes". It follows that expert users of a package like Rubato can wrap up functionality like the best path calculation of the HarmoRubette for pure OpenMusic users in a patch without the help of developers.

5.7 User gains of existing integration components

There are different components of the integration system which are helpful for different users. Because the integration of OpenMusic, Humdrum and Rubato is new, there is no experience yet, how users will accept it. It is reasonable to assume the following use cases:

Computer scientists will use the underlying programming framework for specifying mappings between the visual language and a new target language. It simplifies the development for new mappings, such as OpenMusic to Java, OpenMusic to Perl, etc.

Humdrum users will use the shell mapping component either implicitely by using the evaluation and configuration possibilities of a patch as a front-end for calls to Humdrum commands, or explicitely by modelling complex shell scripts in terms of sets of patches.

Rubato users will implicitely use the FScript mappings to model functions for Rubato hooks.

OpenMusic users will use the functionality of the other packages wrapped by other users into OpenMusic functions.

Since some of the Humdrum and Rubato users will become OpenMusic users, there will be people, who have the know how to do this wrapping. This way these users will also advertise the tools of the other software packages. A joined user community will result, which might stimulate efficiency and creativity and open new research fields.

6 Conclusion

Programming in computer-based music research typically cannot be delegated to professional software developers, either because of lack of funding, or because it is simpler to program by oneself instead of teaching the programmer the application field (music). So both basic programming skills and the knowledge of existing software tools are important conditions for efficient and creative activities in the field of computer-based music research.

While knowledge of existing software is fundamental for efficiency and also stimulating for creativity, the reuse of existing tools often requires too many programming skills. With a visual programming language, OpenMusic has proven to be usable by musicians. The proposed extension of OpenMusic as a frontend for functionality available in other software packages does not require much new expertise for existing OpenMusic users. For other users, learning the visual language is interesting, because there is not only a visual software integration platform, that shields the user from textual programming language details like syntax, but also a set of interesting 'built in' music components that he or she gets for free.

The integrated use of the software packages can best be achieved by integrating the user communities. When discussing different tasks in mailing lists, users from one community can come up with a quick and elegant solution for tasks that are very difficult to handle in another environment. So users can wrap up some functionality and explain to other users, how they can adopt it.

References

AGON, CARLOS and ASSAYAG, GÉRARD (2002). *Object-Oriented Programming in OpenMusic. In: The Topos of Music*, pp. 967–990.

ASSAYAG, G.; C., RUEDA; LAURSON, M.; AGON, C.; and DELERUE, O. (1999). *Computer Assisted Composition at Ircam : PatchWork & OpenMusic. In: Computer Music Journal*, 23(3).

GARBERS, JÖRG (2003). *Integration von Bedien- und Programmiersprachen am Beispiel von OpenMusic,Humdrum und Rubato.* Ph.D. thesis, Fakultät IV – Elektrotechnik und Informatik der Technischen Universität Berlin.

HURON, DAVID (2002). *Music Information Processing Using the Humdrum Toolkit: Concepts, Examples, and Lessons. In: Computer Music Journal*, 26(2):11–26.

KORNSTÄDT, ANDREAS (2002). *Ein Einschubsystem für die musikwissenschaftliche Analyse, basierend auf einem Ansatz zur bruchlosen Modellierung von Anwendungsfamilien mit Rahmenwerken und Komponenten*. Ph.D. thesis, Fachbereich Informatik der Universität Hamburg.

Online references

Rubato:

- The Rubato project: http://www.rubato.org
- FScript download and documentations: http://www.fscript.org

Humdrum:

- Documentation: http://www.music-cog.ohio-state.edu/Humdrum/
- Kern scores: http://kern.humdrum.net/
- Bash manual: http://www.gnu.org/software/bash/bash.html

OpenMusic:

- Short product overview:
  http://www.ircam.fr/produits/logiciels/openmusic-e.html
- User and Developer Guide:
  http://www.ircam.fr/equipes/repmus/OpenMusic/

Rubato:

- Entry page for documentation and references:
  http://www.rubato.org/.

OHR (the OpenMusic, Humdrum, Rubato software integration project):

- Documentation and resources: http://flp.cs.tu-berlin.de/MaMuTh/Ohr/