Declarative Composition and Reactive Control in Marsyas

Jakob Leben
Computer Science
University of Victoria
Canada
jakob.leben@gmail.com

George Tzanetakis
Computer Science
University of Victoria
Canada
gtzan@cs.uvic.ca

ABSTRACT

We present a new coordination language for audio processing applications, designed for the dynamic dataflow capabilities of the *Marsyas* C++ framework. We refer to the language as *Marsyas Script*. It is a declarative coordination language that enables intuitive and quick composition of dataflow networks and reactive processing control. It separates the tasks of dataflow coordination and computation, while increasing the expressivity of the coordination level. This allows more dynamic dataflow behavior and more powerful interaction with other multimedia applications and the physical world. It also increases code portability and allows multiple tools to operate on the same network definition with the purpose of real-time or non-real-time execution, network visualization, operational inspection and debugging, etc. This naturally enhances and extends the functionality within the domain of the Marsyas framework and makes it more accessible to users of other audio software frameworks and languages.

1. INTRODUCTION

There is a growing field of applications that combine sound analysis and synthesis in dynamic ways and frequently in real time. Support for such applications in terms of software is rather fragmented. Audio stream processing in itself is most naturally represented with the synchronous dataflow model of computation (SDF [1]); the most basic form of it with single data rate across the network is implemented in most software frameworks for this purpose. In this model, all data flowing between processing blocks or *actors* are arrays of a single fixed size. However, algorithmic music creation, but also sound analysis, typically require expressivity beyond the static single-rate model. Frameworks that focus on sound synthesis and music creation (e.g. SuperCollider [2], ChucK [3] [1], Pure Data [4], Max/MSP, ...) typically maintain the single-rate SDF model. On the other hand, to provide expressive musical control over sound, they implement powerful means of manipulating sparser and variable-rate streams of control data.

[1] ChucK is unique in operating on the single-sample dataflow level.

In contrast, one common property of sound analysis applications is that they operate on streams that span different information domains: e.g. time domain (audio), frequency domain (spectrum), statistical summarization of information streams, etc. This requires more expressive dataflow models: streams of different data formats and rates. Moreover, processing itself is sometimes affected by information produced as the result of analysis; the dataflow structure must adapt to the content of the streams being processed (e.g. detected sound onsets). There has been very little support for integration of such multi-rate and dynamic dataflow characteristics with easy-to-use frameworks for sound synthesis and music creation.

The formalization of multi-rate and dynamic dataflow models has mostly taken place in the signal processing community (for a comprehensive overview see [5]) with applications mainly dealing with the encoding and decoding of complex multimedia streams. Elaborate formalisms are most frequently employed on embedded systems, but multimedia applications on general-purpose systems also borrow some of the techniques. Despite an abundance of formalisms and concrete applications, few solutions are provided in form of abstract frameworks for general purpose computing machines and accessible to a wider, less technical user community or with focus on computer music.

Marsyas [6, 7, 8] was one of the first frameworks to provide multi-rate and constrained dynamic dataflow capabilities targetting specifically sound analysis applications, with a user-friendly programming interface on the level of dataflow coordination as well as efficient implementation on the level of internal actor computation. It is based on the well-established C++ language and running on general-purpose computing machines. In this paper, we present Marsyas Script - a new coordination language that greatly simplifies existing workflow as well as facilitates further aspects of dataflow coordination. It makes powerful sound analysis even more accessible to less-technical audiences and increases flexibility required for interfacing with other sound synthesis and computer music frameworks.

2. CONTRIBUTIONS

Coordination of dataflow networks in Marsyas was previously done in imperative languages (C++ or through Python bindings), which made the structure of networks barely apparent from code used to specify them. Marsyas Script is a completely declarative language. Readability of code is greatly improved in the new language, especially because the hierarchical structure of code directly corresponds to the hierarchical network composition, and the dataflow-specific concept of *scope* simplifies addressing of different parts of a network (see section 5.6).

Marsyas Script provides expressive programming of reactive control flow, regardless of whether information originates from the external world - Open Sound Control (OSC) [9] messages, graphical user interfaces, etc. - or internally - as a result of data analysis. This gives more power over the dynamic dataflow capabilities into the hands of the user, as well as facilitates more rich interaction with other applications. Another benefit of a high-level coordination language is reliance on powerful and efficient functionality implemented in the host language (C++). The large number of audio processing algorithms provided by the Marsyas C++ library is accessible as dataflow actors. The performance is just as efficient as if dataflow coordination was expressed in C++.

Moreover, code translation is quick and performed on-the-fly at application start-up. This increases portability of network definition code between machines, users and applications. It allows a number of precompiled tools to operate on the same network definition. We have implemented a generic executable that instantiates and runs any network defined in the new languages, either for real-time or non-realtime audio processing similar to an audio plugin host but with more extensive functionality. Another application allows inspection and debugging of dataflow by visualization of network structure, step-by-step execution and selective plotting of intermediate stages.

3. RELATED WORK

In spite of the dataflow model of computation having a straightforward visual representation and yielding many visual programming frameworks (such as Pure Data [4], Clam [10], Max/MSP, NI Reaktor, to name just a few most popular in the domain of sound processing), textual programming for sound processing is not only persistent (for example SuperCollider [2], ChucK [3]), but new languages and frameworks keep being created, indicating that textual programming has its own merits among which the combination of code expressivity and brevity is probably the major factor.

A contemporary trend is the resurrection of fine-grained dataflow programming. It allows compliation into optimized code for today's and tomorrow's computing devices with modest support for coarse-grained task parallelism and increasing support for massive fine-grained data parallelism. Examples are Faust [11] and more recent Kronos [12, 13] for audio, as well StreamIt [14] for general-purpose stream processing. In contrast, the Marsyas Script is a rather coarse-grained coordination language and its strength remains reusability of the large collection of low-level algorithms already developed in the Marsyas C++ library.

ESSENTIA [15] is another recent C++ library of algorithms specifically for audio analysis and music information retrieval. Akin to Marsyas, algorithms are embedded in a dataflow actor-like interface, they are C++ templates, so they accept any input and output data type as parameter and vector data may be of any size, which supports equivalent dataflow flexibility as Marsyas. However, there is only a basic support for composition of actor networks, there is no concept of control flow and no support for reactive coordination of actors.

The syntax and reactive control in Marsyas Script were inspired by the QML language which is part of the Qt framework for graphical user interface development. QML is also a declarative language where graphical items are composed hierarchically in a similar syntactical fashion and properties of items are bound to reactive expressions involving properties of other items.

It is worth noting that such syntax for hierarchical composition could not be used to define a dataflow network without the concept of implicit patching as introduced previously in Marsyas [7]. A similar functionality of implicit patching is present in Faust [11], although there it does not rely on hierarchical composition, but patching is rather performed by binary composition operators.

There has been previous work on expressive control flow in Marsyas [16] but only in reaction to explicitly scheduled timed events, rather than implicit changes of control values. There was no concept of reactive *bindings* as defined here in section 5.4. The paper [16] also mentions a "Marsyas Scripting Language (MSL)" featuring declarative network composition similar to Marsyas Script presented here, but to the best of our knowledge, that work has never been completed, and the MSL only supports limited imperative programming.

Our temporal operators for reactive control flow (see section 5.4) are inspired by declarative synchronous languages LUSTRE and SIGNAL [17] where similar operators exist to filter one sequence of events to those that match occurences of events in another sequence. However, in contrast to these languages where sequences only have values at transient moments in time, our control flow is rather similar to values in the imperative language ESTEREL [17] which persist and are observable across time while only their change is discrete. The temporal operators thus sample values from one stream at arbitrary moments as if it was a continous step function of time.

Our control alternative (the `when` statement, 5.5) is inspired by the concept of *states* in QML. It serves to change larger-scale behavior depending on events. This is similar to composition of behaviors and events in the Functional Reactive Programming paradigm (FRP [18]). Since controls can result in changes of output rates and formats of actors, enabling and disabling actors, or potentially even instantiation of new actors, this may also facilitate a particular form of dynamic dataflow where the network structure is controlled by a higher level state machine.

4. MARSYAS ARCHITECTURE

The concepts in Marsyas Script correspond to a large degree with the concepts of the Marsyas C++ framework which provides the underlying implementation. Dataflow actors are called *MarSystems* - they represent basic building blocks for sound analysis and synthesis algorithms: audio file and real-time input and output, feature extraction, statistics, sine and noise generators, filters, etc.

To support different dataflow rates, input and output data of any MarSystem is a matrix (2D array) of arbitrary dimensions, where columns represent successive information in time, and rows represent concurrent information, possibly originating from different producers or distributed to different consumers. All input and output is packed into such data structures. Hence all input and output of an individual MarSystem is of equal rate (respectively), but MarSystems in a network may have different rates. Each MarSystem has an intrinsic relation between its input and output rates, but can operate at any mutliple of this ratio.

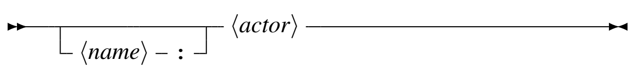
MarSystems compose hierarchically. Connection among siblings are established using the implicit patching paradigm [7]: instead of explicitly connecting each input and output, there is a special class of *composite* MarSystems which connect their children in specific patterns. Examples include: serial composition, parallel composition (composite input is split among children) fanout (entire input is passed to each child), etc. Data rates among connected siblings must match, hence they are automatically propagated from outputs to inputs. Rates may change across hierarchical levels: special composites execute their children multiple times within each of their own execution, each time passing successive chunks of input and accumulating output. We have found this to be an architecture flexible enough for a variety of audio synthesis and analysis scenarios, while allowing efficient implementation on general-purpose computers, even in the case of dynamically changing rates.

Each MarSystem has a set of parameters named *controls*. The *controls* can be changed and observed asynchronously with respect to the dataflow execution. They may control any aspect of a MarSystem's operation, including its input and output data rates. They are also used for output of sporadic information, typically resulting from analysis (detected pitches and onsets for example). Controls can be linked, so that when one produces new information all others are updated. This is the foundation for the expressive reactive control described in section 5.4.

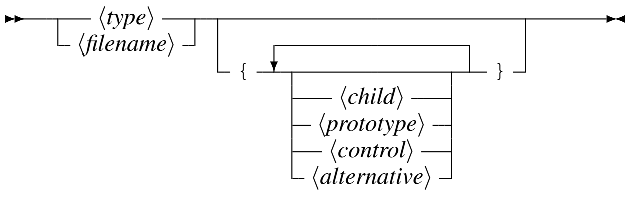
5. SYNTAX AND SEMANTICS

This section gives an overview of the syntax. Only high-level grammatical symbols are described. In particular, the terminal symbols (numbers, strings, identifiers, etc.) are not decomposed, as they follow usual lexical conventions. The reader is invited to see online Marsyas documentation for a detailed exposition.

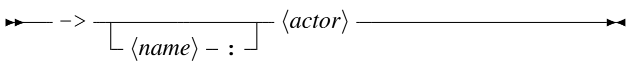
⟨*script*⟩



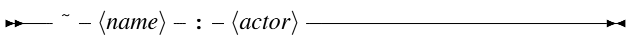
⟨*actor*⟩



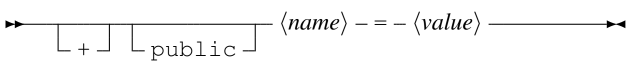
⟨*child*⟩



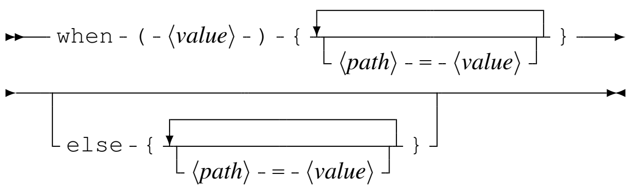
⟨*prototype*⟩



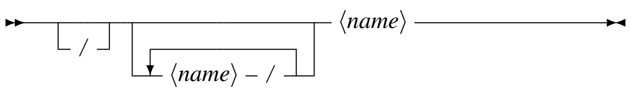
⟨*control*⟩



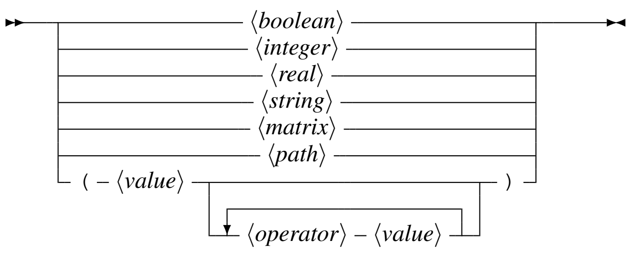
⟨*alternative*⟩



⟨*path*⟩



⟨*value*⟩



5.1 Script

A script consists of a definition of a single top-level actor which, by recursion, will include the definitions of all its children. The top-level actor may optionally be assigned a name, otherwise a default name "network" will be assigned automatically.

5.2 Actors

An essential component of an actor definition (the ⟨*actor*⟩ symbol) is either a type name (corresponding to the C++ class name of a MarSystem), or a filename of another script, in which case the entire network defined in that script is used as a prototype.

The body of an actor definition within curly braces specializes the definition of its type by assigning values to controls, adding new controls and adding actor instances as children.

A child actor instance (the ⟨*child*⟩ symbol) is introduced using the arrow symbol, which creates an intuitive association with the flow of data between parents and children or between siblings. A child may optionally be named, which allows it to be addressed and also plays a role in name lookup of other actors (see 5.6). Following the child's name is its definition, a recursion of the syntax of its parent definition. Syntactical hierarchy thus reflects actor network hierarchy.

5.3 Controls

A control declaration (the ⟨*control*⟩ symbol) operates on controls of the enclosing actor definition. It can either assign values to existing controls or create new controls (if prefixed with the `+` symbol).

Some controls are defined by C++ implementations of MarSystems and those directly affect MarSystem operation or report results of its computation or changes of its internal state. A new control may function as a named result of a control computation to be reused in other control expressions. Another use is to make control values deeper in the hierarchy more accessible to the outside world by assigning them to new top-level controls.

When assigning values to existing controls, the type of the value must match the type of the control. When creating new controls, their type becomes that of the assigned value.

Controls (either pre-existing or new) may also be declared *public* (by prefixing the name with the `public` keyword), which at the moment of this writing has no effect on the internal operation of the script, but it affects how controls are treated by external tools: for example the graphical inspector application optionally hides all non-public controls.

5.4 Control Values and Reactive Expressions

A control assignment is a *binding* between a control and the value of a reactive expression. An expression is a composition of literal constant values, control paths (5.6) denoting their changing values over time, and operators on those values. Whenever any constituent value changes, the expression is reevaluated and the value of the bound control is updated.

Each value has one of the following types, inherited from the Marsyas C++ framework: integer or real number, 2D matrix of real numbers, boolean, string. An expression also has a type: that of its result value, as defined intrinsically by operators and types of operands.

The following arithmetic operators are defined on any pair of numbers or equal-size matrices: `+`, `-`, `*`, `/`. On matrices, they operate point-wise. The following arithmetic comparison is defined on numbers: `<`, `>`, `<=`, `>=`. The following comparison is defined among pairs of numbers, pairs of equal-sized matrices, or pairs of any other identical types: `==`, `!=`.

There are two special *temporal* operators: `on` and `when`. The `on` operator produces a value that becomes the current value of the left-hand-side operand whenever the right-hand-side value changes. The `when` operator does the same, but only when the boolean right-hand-side operand becomes `true`.

5.5 Control Alternative

The declaration of a *control alternative* (the ⟨*alternative*⟩ symbol) consists of a condition in form of a boolean expression and two sets of control assignments separated by `else`. Whenever the value of the condition changes, either the first or the second set of bindings will be activated, depending on whether the new value of the condition is `true` or `false`, respectively. Note that the `else` part is optional, in which case no change of control bindings will happen when condition switches from `true` to `false`.

5.6 Name Scope and Path Resolution

Controls in control expressions (5.4) may be those belonging to any *named* actor in the network. They are addressed using control paths (the ⟨*path*⟩ symbol). A path consists of a name of a control optionally prefixed by a sequence of actor names in a hierarchical order from the control owner up.

A path always has an actor as an implicit *origin*; the first name in the path denotes a *child* actor of the origin. The `/` at the beginning of a path makes a path *absolute*, so regardless of where it is used it's origin is always the top-level actor. The top-level actor's name is actually never used - it is represented by the initial `/`. Paths starting with a name are *relative* paths originating at the actor definition within which they are used.

A path may address controls across hierarchy without the need for all the actors on the way to be named. This is enabled by the concept of *scope*. Each named actor (5.2) is a scope which contains names of those descendants that are hierarchically separated from the scope actor only by unnamed actors, regardless of how hierarchically remote they are. In addition, the root actor is considered a scope, regardless of whether it is named. Names of actors must be unique within their enclosing scope. Thus, a path is a sequence of named actors where each following actor is in the previous one's scope but not necessarily a direct child.

An absolute path also conforms to the Open Sound Control message address specification [9], which provides an immediate addressing solution for communication with the outside world.

5.7 Prototypes

A prototype declaration (the ⟨*prototype*⟩ symbol) allows an actor definition to be used as a new actor type and thus instantiated multiple times. It is introduced using the `~` symbol, followed by a name for the new type, and its definition (the ⟨*actor*⟩ symbol).

Any actor definition (including prototypes themselves) may use prototypes already defined in the same or enclosing actor definitions. On the other hand, a prototype definition is an isolated scope (5.6), preventing its nested child *instances* to be addressed from enclosing code and the other way around - unless it is instantiated and given a name.

6. EXAMPLES

A complete example of typical usage of Marsyas Script and framework is provided in figure 1. The script implements a basic onset detection algorithm consisting of peak picking from an onset detection function defined as spectral flux.

The top-level controls `inSamples` and `israte` specify the desired audio block size and sampling rate at the root of the dataflow graph - all downstream actors automatically adjust their input and output formats. The `AudioSource` will open a connection to the default audio device for real-time audio acquisition and automatically re-block audio as specified above. The `ShiftInput` produces an overlapping sequence of audio windows with the hop size of input amount of samples and window size specified with the `winSize` control.

The stream of overlapping audio windows is forked into two branches: one computes signal energy and delays it to temporally match the onsets detected in the other branch which requires inspection of past and future windows to determine whether a window is an on onset.

The additional top-level control `onset` is defined as the normalized audio energy at times of onsets. This is enabled by combining the conversion of the energy value from data flow to control flow by the `FlowToControl` and the boolean control value of the `onsetDetected` control of the `PeakerOnset`, which becomes `true` after an audio window is detected to contain an onset and `false` otherwise. Using the `when` temporal operator, the resulting value will only change at times of onsets.

By executing the script in real-time using the `marsyas-run` tool provided by the Marsyas framework, it is possible to have OSC messages automatically sent for every change of the top-level `onset` control, to an arbitrary destination using UDP/IP. This allows for convenient interfacing with other audio applications, most typically for the purpose of audio synthesis.

For comparison, see figure 2 which shows the same algorithm implemented in C++ in a manner most typical before the introduction of Marsyas Script. To attempt better readability, C++ code usually consists of three distinct steps, the first one being actor composition, followed by control setting and linking, and finally a run loop which continously invokes an iteration of dataflow processing. Due to limited reactive capabilities, the run loop usually contains explicit imperative inspection of control values and computation of new ones.

The example in Marsyas Script (fig. 1) clearly shows improved code expressivity and brevity over the C++ example (fig. 2). Dataflow structure is much more apparent. Code is not cluttered with unnecessary syntactical features of C++. Moreover, control addressing is much simpler because of locality of assignments within actor definitions (5.3) and sophisticated remote control path resolution (5.6). Finally, interfacing with the outside world is simple using reactive expressions to define outside-facing controls.

```
Series
{
  + public onset =
      ( energy_out/value / energy/inSamples
        when onsets/onsetDetected )

  inSamples = 512
  israte = 44100.0

  -> AudioSource
  -> ShiftInput { winSize = 1024 }
  -> Fanout
  {
    -> Series {
       -> energy: Energy
       -> DelaySamples{delay=4}
       -> energy_out: FlowToControl
    }
    -> Series {
      -> Windowing -> Spectrum -> PowerSpectrum
      -> Flux { mode = "Laroche2003" }
      -> Memory { memSize = 25 }
      -> onsets: PeakerOnset {
        threshold = 6.5
        lookAheadSamples = 4
      }
    }
  }
}
```

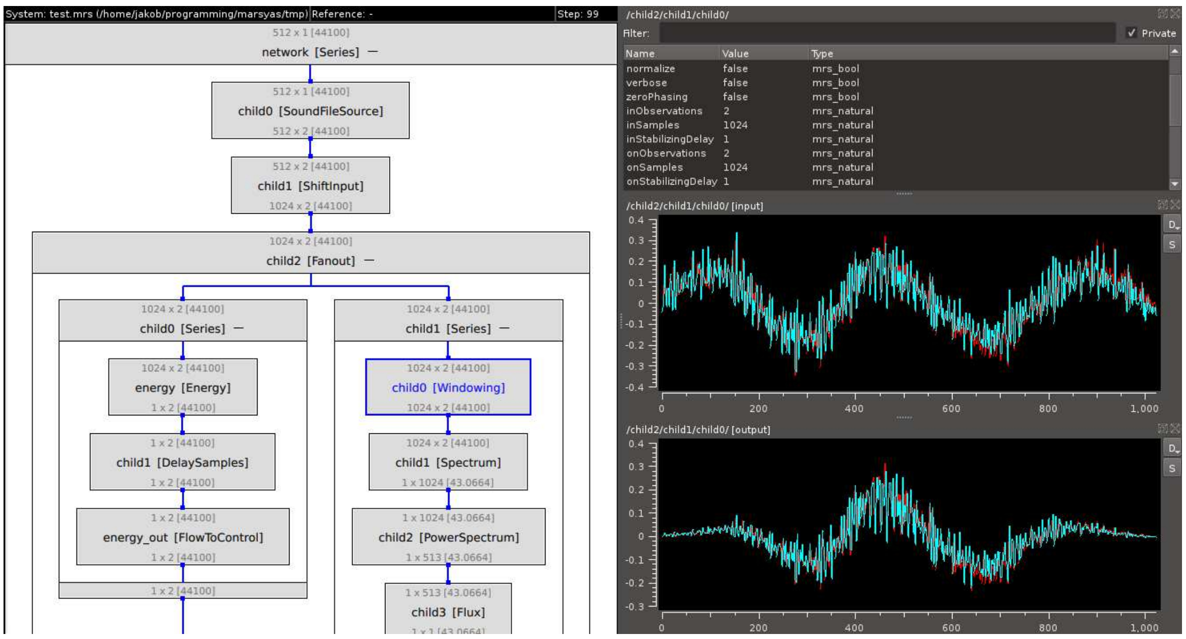
Figure 1. Real-time onset detection example in Marsyas Script. Note the clarity of structure and data flow, code brevity and expressive control flow (the top-level 'onset' declaration).

7. RELATED TOOLS

Along with the Marsyas Script language a set of programs is being developed that operate on scripts. The most important one to make scripts useful is `marsyas-run`, which takes a script file as argument, instantiates the dataflow network defined in the script, and runs the network in real-time (if real-time audio input or output is declared in the script) or as fast as possible with the purpose of processing audio files. In any case, it will stop processing when the boolean top-level control `done` becomes `true`, which can be used to signal the end of an input file by binding to a control of a `SoundFileSource`. Moreover, control values can be set by the user at start of the program using its arguments, which allows a single parameterized script to be used in different scenarios.

The `marsyas-run` program also implements OSC communication using UDP/IP, and can be instructed to send changes in controls as OSC messages, as well as apply incoming OSC messages to control values. Since control paths conform to the OSC address specification, there is a direct mapping between the two concepts.

Another graphical application that helps in script development as well as MarSystem implementation is the Inspector. It interprets a script and provides a visualization of the dataflow network structure. It also allows executing the dataflow iteration by iteration and inspecting all control values and all data flowing between actors. It is a great tool for verification and debugging in two aspects: the aspect of script code as well as the aspect of system integration of C++ MarSystem code. Figure 3 shows a screenshot of the Inspector used with a modified version of the script given in figure 1 which sources data from an audio file.



/child2/child1/child0/

Filter: ✓ Private

Name	Value	Type
normalize	false	mrs_bool
verbose	false	mrs_bool
zeroPhasing	false	mrs_bool
inObservations	2	mrs_natural
inSamples	1024	mrs_natural
inStabilizingDelay	1	mrs_natural
onObservations	2	mrs_natural
onSamples	1024	mrs_natural
onStabilizingDelay	1	mrs_natural

Figure 3. Marsyas Inspector displaying a dataflow network, a list of control values and two plots of data at different stages of processing (before and after applying the windowing function).

```
MarSystemManager mng;

//// 1. Compose
// ...Create a composite and add processing actors:
MarSystem *energy_branch =
    mng.create("Series", "energy_branch");
MarSystem *onset_branch =
    mng.create("Series", "onset_branch");
// ...
energy_branch->addMarSystem
    (mng.create("FlowToControl", "energy_out"));
onset_branch->addMarSystem
    (mng.create("PeakerOnset", "onsets"));
//... Create and fill root composite
MarSystem *network = mng.create("Series", "net");
network->addMarSystem
    (mng.create("AudioSource", "input"));
//...

//// 2. Configure
network->updControl("mrs_natural/inSamples", 512);
onset_branch->updControl
    ("PeakerOnset/onsets/mrs_real/threshold", 6.5);
//...

//// 3. Run
MarControlPtr onset_control =
    onset_branch->getControl
    ("PeakerOnset/onsets/mrs_bool/onsetDetected");
MarControlPtr energy_control =
    energy_branch->getControl
    ("FlowToControl/energy_out/mrs_real/value");
while(should_run)
{
  network->tick();
  bool onset_detected = onset_control->to<bool>();
  if (onset_detected)
  {
    mrs_real energy =
        energy_control->to<mrs_real>() / 1024.0;
    // send "energy" as OSC ...
  }
}
```

Figure 2. Real-time onset detection example in C++, showing 3 typical distinct steps (compose, configure, run). The example is incomplete - parts have been left out for brevity.

8. CONCLUSIONS AND FUTURE WORK

We have shown how the new coordination language named Marsyas Script builds on top of the functionality of the Marsyas C++ framework with features specific to sound analysis applications (multi-rate and dynamic dataflow). The declarative nature and expressive reactive programming features make the previous functionality more accessible to a variety of users end further empowers them to easily express complex dynamic processing control. In combination with the built-in OSC communication capability in the `marsyas-run` program, reactive control expressions provide flexibility to effectively interface with other audio software. Marsyas Script allows development of generic tools to inspect and debug dataflow networks of which the Marsyas Inspector is an example. In general, it makes network definition code more portable across applications, machines and users.

There is a broad area of research interesets and future development that the work presented in this paper inspires.

One particular issue we would like to address is synchronization of dataflow and control flow. Although control flow in Marsyas is synchronous in itself, it is asynchronous with respect to dataflow actor firings: control changes may be arbitrarily interleaved with actor firings. This becomes an issue when multiple actors depended on the same control value and a change in value occurs in the middle of a dataflow processing iteration - they may take the control information into account either in the current or the next iteration. StreamIt addresses this issue with the concept of *information wavefront* [14] which allows synchronization of out-of-stream message delivery with a particular actor firing. Another solution would be to associate a timestamp with each control change and use it to precisely synchronize the application of change with data flow. However, the dataflow model in the most general sense (and in particular in multi-rate scenarios) has no well-defined association of flow data with time and duration (only data order is defined), so enhancing dataflow with strict temporal semantics would be a prerequisite for usage of timestamps for synchronization.

Another application of more strict temporal semantics would be synchronization of distributed systems in a manner transparent to the user, so that temporal operators could be used on all streams equally, regardless of how remote their origin is. Conceptually, this would be an attempt at a combination of synchronicity in languages for reactive systems [17] with multi-rate dataflow.

Marsyas Script could also enable simplification of automated unit and integration testing. No matter how small functionality is tested, each unit test typically requires code to set up and clean up the testing environment. Writing this code repeatedly with only slight modifications for each new test is a tedious endeavour. Instead, scripts could be used as parameterized and dynamic testing environments where individual units would be easily plugged in. The size of this environment could be minimal, for pure unit testing, or a complex dataflow network, for integration testing.

We are also interested in translation of audio stream processing defined in expressive and intuitive languages into forms maximally optimized for efficient execution, including parallelization. However, this most likely calls for work on new foundations beyond those that the current Marsyas C++ framework provides.

Acknowledgments

The authors gratefully acknowledge the support of the National Science and Engineering Research Council of Canada. We would also like to thank the many Marsyas developers and users who have influenced and tested the ideas described in this paper.

9. REFERENCES

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proc. IEEE*, 1987, pp. 1235–1245.

[2] J. McCartney, "Supercollider: a new real time synthesis language," in *Proc. Int. Computer Music Conf.*, 1996.

[3] G. Wang, "The chuck audio programming language. a strongly-timed and on-the-fly environ/mentality," Ph.D. dissertation, Princeton University, 2008.

[4] M. Puckette, "Pure data," in *Proc. Int. Computer Music Conf*, 1997.

[5] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of signal processing systems*. Springer Science & Business, 2013.

[6] G. Tzanetakis and P. Cook, "Marsyas: A framework for audio analysis," *Organised sound*, vol. 4, no. 3, pp. 169–175, 2000.

[7] B. Stuart and G. Tzanetakis, "Implicit patching for dataflow based audio analysis and synthesis," in *Proc. Int. Computer Music Conf*, 2005.

[8] G. Tzanetakis, "Marsyas-0.2: a case study in implementing music information retrieval systems," in *Intelligent Music Information Systems*. IGI Global, 2007.

[9] A. Schmeder, A. Freed, and D. Wessel, "Best practices for open sound control," in *Proc. Linux Audio Conf.*, 2010.

[10] X. Amatriain, P. Arumi, and D. Garcia, "Clam: A framework for efficient and rapid development of cross-platform audio applications," in *Proc. 14th ACM Int. Conf. on Multimedia*, 2006.

[11] Y. Orlarey, D. Fober, and S. Letz, "Faust: an efficient functional approach to dsp programming," in *New Computational Paradigms for Computer Music*. Delatour France, 2009.

[12] V. Norilo and P. Rautatiekatu, "Introducing kronos-a novel approach to signal processing languages," in *Proc. Linux Audio Conf.*, 2011.

[13] V. Norilo, "Recent developments in the kronos programming language," in *Proc. Int. Computer Music Conf.*, 2013.

[14] S. Amarasinghe, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *International Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 261–278, 2005.

[15] D. Bogdanov, N. Wack, E. Gmez, S. Gulati, P. Herrera, O. Mayor, G. Roma, J. Salamon, J. Zapata, and X. Serra, "Essentia: an open-source library for sound and music analysis," in *Proc. 21st ACM Int. Conf. on Multimedia*, 2013, pp. 855–858.

[16] N. Burroughs, A. Parkin, and G. Tzanetakis, "Flexible scheduling for dataflow audio processing," in *Proc. Int. Computer Music Conf*, 2006.

[17] N. Halbwachs, *Synchronous programming of reactive systems*. Springer, 1992.

[18] C. Elliott and P. Hudak, "Functional reactive animation," *ACM SIGPLAN Notices*, vol. 32, no. 8, pp. 263–273, 1997.