MWSCCS: A Stochastic Concurrent Music Language

BRIAN J. ROSS
Brock University
Department of Computer Science
St. Catharines, Ontario, Canada L2S 3A1
bross@sandbanks.cosc.brocku.ca

Abstract

The paper describes a music composition language MWSCCS – the Musical Weighted Synchronous Calculus of Communicating Systems. MWSCCS is a stochastic language based on Tofts' WSCCS and Milner's SCCS algebra. Main features of MWSCCS are its straight-forward approach to probabilistic execution, the ability for processes to generate musical events autonomously or to communicate amongst each other, the ability to write prioritized reactive processes, and its concise, hierarchical set of operators. MWSCCS diverges from the pure WSCCS algebra by introducing various devices useful for music, for example, a MIDI mappable event space, and stream repetition for the generation of repetitive melodies and rhythms. MWSCCS's execution engine consists of a search mechanism, which replaces the pure algebra's denotational semantics over streams. The implementation of the language benefits from the formal semantic definition of the original process algebra. The language is implemented in Prolog. Work is under way in creating compositions with MWSCCS that exhibit interesting chaotic and self-organizing behaviours.

1 Introduction

The perspective of music as a concurrent activity is well established. Music models using Petri nets (Haus & Rodriguez, 1988; Haus & Sametti, 1992) and algebraic interleaving (Chemillier & Timis, 1988; Chemillier, 1990) are rooted in the concurrent computation paradigm to different extents. This papers contribution is to suggest a music programming language based on a process algebra. The process algebra model considers music as comprised of discrete, concurrent behaviors that can be observed and reacted upon by musical processes. This can be intuited by considering a small improvisational jazz band. A competent musician does not play independently with respect to the rest of the band, but rather, reacts to musical events and cues from the other musicians. Such cues can range from maintaining step with subtle tempo drifting, to compositional cues such as the playing of notes that fulfil melodic or harmonic obligations with the notes played by other instrumentalists. We therefore identify a *musical event* as being an observable discrete phenomena that can be observed by others, and which can ellicit reactions. We identify the ability to observe, recognize, and react to such events as being an important ingredient to human and computer music processing and composition.

The music programming language discussed in the paper is the Musical Weighted Synchronous Calculus of Communicating Systems, or MWSCCS. It is an implementation of the MWSCCS process algebra, enhanced with features useful for music composition. Some benefits of the language include a concise yet powerful set of basic primitives, the ability to create abstract hierarachical musical systems, and an intuitive mechanism for specifying complex stochastic behaviours of musical processes.

The format of the paper is as follows. A brief discussion of the process algebra foundation of MWSCCS is in section 2. Section 3 discusses the MWSCCS language, and section 4 gives some example MWSCCS compositions. A discussion concludes the paper in section 5.

2 Process Algebra

An *agent* or *process* is an abstract mechanism whose behavior is characterised by discrete actions. Process algebra are mathematical formalisms for modelling processes. There are many process algebra in the literature (eg. (Hoare, 1985; Hennessy, 1988; Milner, 1989)), each of which establish different perspectives and styles for modeling concurrency. Process algebra are effective models of concurrency because of the high degree of abstraction possible with them, as well as their intuitive "programming language" feel. Along with process algebra, two other algebraic models of concurrency include Petri Nets (Peterson, 1977) and trace theory (Aalbersberg & Rozenberg, 1988). Although all three formalisms share much in common – they all define finite automata – they do differ in the nature of their semantics and subsequent analyses. For example, process algebra tend to be more abstract and specification-oriented than Petri nets, while the latter describe concurrency at a more intricate structural level.

The MWSCCS process algebra is described in detail in (Ross, 1995a; Ross, 1995b). MWSCCS is a musically-extended version of Tofts' WSCCS algebra (Tofts, 1990), which is in turn a stochastic version of Milner's SCCS (Milner, 1989). SCCS (synchronous calculus of communicating systems) is a process algebra in which processes contribute their visible activity synchronously, or in other words, in unison with a global clock. The algebra also contains operators for structuring process definitions, renaming and inhibiting actions, and permitting nondeterministic choices of behaviour. WSCCS adds to SCCS a mechanism useful for stochastic and reactive control of nondeterministic choice. Finally, MWSCCS adds to WSCCS some denotations useful in a musical context, for example, an event space mappable to MIDI. The algebraic semantics of all these algebra are defined in terms of transitional rules of inference. Each rule describes the behaviour of an operator in terms of the relation it defines with respect to a transition over the stream of observable events. A discussion of these rules is beyond the scope of this paper (please see the aforementioned references). However, it is worth noting that a transitional inference rule semantics is invaluable for creating a correct implementation for the algebra as done in this research.

MWSCCS treats music as a stream of observable discrete events. When applied to music, this stream denotes the horizontal structure of music – the relative sequential order of events with respect to each other. The vertical component of music – event simultaneity – is naturally denoted by the event domain of multi-particle actions. A *particle* is the smallest visible atomic event definable . Multiple particles can coalesce to produce an *action*. When multiple processes synchronously communicate their actions, their constituent actions are combined to form new actions. The process $a\overline{b}.cd$ represents two actions, $a\overline{b}$ followed by cd. Particles have two polarities (eg. a and $\overline{a}$), which cancel themselves if they

coincide in one action. '1' is the identity action, and represents silence. '0' represents termination. Then the expression

$$a.b.c.d.0 \;\#\; a\bar{b}.cd.1.0 \;\#\; ab.\bar{d}\bar{e}.f.0$$

represents three processes communicating concurrently via the composition operator #. The expression reduces to $aa\bar{b}ab.bcd\bar{d}\bar{e}.c1f.d00$, which in turn simplifies to $aaa.bc\bar{e}.cf.0$. Note how the d in the first term is not generated, because the other two processes have already terminated. The power of this representation is that it naturally models musical activity. Each particle above can be considered to be a musical note, and composite notes in turn denote chords. We can colour the notes to denote particular voices or instruments if desired. A ramification of this musical interpretation is that a whole composition quits as soon as one process (instrument, voice, musician) quits.

Another useful feature of the algebra, inherited from WSCCS, is its representation of stochastic processes. In the choice expression,

$$2\omega^2 ab \;+\; 3\omega^2 cd \;+\; 4\omega^1 ef.g \;+\; 5\omega^1 h$$

the first two terms are considered before the last, as their priority value ω^2 is higher than ω^1. The first and second terms are selectable with probabilities of $2/(2+3) = 2/5$ and $3/5$ respectively, and the last terms with probabilities $4/9$ and $5/9$ respectively.

3 The MWSCCS Language

3.1 The event space

Section 2 introduced particles and actions, and how they coalesce and cancel during synchronous communication. MWSCCS supports the following event space $\mathcal{A}$:

$$\mathcal{A} \;=\; \{\; \textit{Generic Actions} \;\} \;\cup\; \{\; \textit{Music Notes} \;\} \;\cup\; \{\; \mathbf{1},\, \surd,\, -\surd \;\}$$

Generic actions are vanilla process algebraic communications, and normally are lower-case constants. *Musical Notes* uses notation for identifying standard 12-semitone multi-octave notes. For example, *as5* is *A-sharp octave 5*. In addition, notes can be coloured with a channel (between 1 and 16) and velocity (between 0 and 255), as in *as5 ch 3 vel 155*. In order to map to MIDI events, note actions can be interpreted as *Note On* and *Note Off* messages. In this interpretation, $as5$ and $as5\dagger$ denote *A-sharp On* and *A-sharp Off* respectively. The action **1** denotes silence. All actions (except **1**) have positive and negative polarities, for example, a and $-a$. The reserved particle $\surd$ denotes "active termination" of a process. This termination differs from absolute termination (**0**) in that the process does not end, but remains silent throughout the rest of the composition, having become equivalent to a process *Silent*:

$$Silent \;=_d\; \mathbf{1}.Silent$$

A process that emits $\surd$ just before becoming silent is called *well-behaved*.

Composite actions are denoted by tuples. The term $(c3,\ e3,\ g3)$ represents the chord C-major. The language permits simple function expressions over particles, which is useful in concert with parameter passing. The term $f(X+2)$ denotes the musical note two semitones above note X.

Probability and priority codes may prefix terms. In $(F,P)\Delta(Action)$, F is the relative frequency and P is the priority. If P is missing as well, the priority is taken to be 1. If F is missing, then F is 1. The expression

$$(2,2)\Delta(as5,\ -bf)\ +\ (3,2)\Delta(a5,\ -bf)\ +\ (4,1)\Delta(c5)\ +\ (5,1)\Delta(d5)$$

therefore treats the first two terms as higher priority than the other two. Probabilities are assigned as described in section 2.

3.2 Operators

The language $\mathcal{E}$ of agent expressions over an event space $\mathcal{A}$ is defined by the grammar

$$\begin{array}{rcl}\mathcal{E} & ::= & \mathbf{0}\ \mid\ X\ \mid\ (F,P)\Delta\alpha.E\ \mid\ E\lceil A\ \mid\ E\backslash A\ \mid\ E[f]\ \mid\ E_1+E_2\ \mid\ E_1\#E_2\ \mid \\ & & E_{label}(\tilde{t})\ \mid !E\ \mid\ N\ rep\ E\ \mid\ inf\ rep\ E\ \mid\ (N)\end{array}$$

where $E, E_i \in \mathcal{E}$; F, P, N are integers > 0; A are particles; α is an action; f is a particle renaming function; $\tilde{t}$ is a parameter list; and E_{label} is a process name. Processes definitions are defined by:

$$E_{label}\ \ =d=\ \ E$$

The *null* process $\mathbf{0}$ is the process that has absolutely terminated. The *prefix* operator in $\alpha.E$ represents the process that can perform the action α, thereafter becoming the process E. The expression $\alpha.\mathbf{0}$ abbreviates to α.

The *permission* operator in $E\lceil A$ denotes the process that performs the particles that are members of the set A. In effect, the permission operator prunes all actions not in A. The *restriction* operator in $E\backslash A$ is the converse of permission, except that it lists the actions which cannot be generated by E. In fact, $E\backslash A\ =\ E\lceil(\mathcal{A}-A)$.

The *relabelling* operator in $E[f]$ renames particles according to f, while leaving weights alone. For example, $E[b/a]$ replaces particle a with b in each action emitted by E.

The *choice* operator E_1+E_2 represents the choice of execution of a set of processes according to priority and probability prefixes on the terms, as described above.

The *parallel composition* operator in $E\ \#\ F$ forces concurrent processes E and F to synchronously communicate with one another. We let $E\ \#\ F\ =\ F\ \#\ E$. If both E and F have no probability or priority information, then a new action is formed by their combined actions:

$$a.E\ \#\ b.F\ =\ ab.(E\ \#\ F)$$

When prefixes are involved, rules from the MWSCCS algebra describe how to combinate them to create new prefixes for the results.

Wherever a process variable X is found, the $\mathcal{E}$ expression bound to it is used. A reference to a process E_{label} defined with an expression $= d =$ causes that process definition to be invoked, possibly with parameter passing.

$!E$ denotes the infinite invocation of an expression. E is invoked until it absolutely terminates, at which time it is reinvoked. On the other hand, $inf \;\; rep \;\; E$ invokes E until it terminates, and then infinitely repeats the *stream* generated by E. Therefore, $!E$ generates in an indefinite stream of varying behaviour, while $inf \;\; rep \;\; E$ gives an indefinite stream of repetitive behaviour.

Finally, the notation (N) represents a delay of N clock ticks. For example (4), is equivalent to **1.1.1.1**. Because compositions usually require lengthy pauses between notes or between note on/off messages, this notation is very convenient.

3.3 Useful Extensions

Sometimes it is useful to model asynchronous processes. Unlike the processes above, asynchronous processes may nondeterministically wait or stall before elliciting their observed behaviour. We can represent an asynchronous process by the following meta-operator:

$$\bullet(P) \;\; = d = \;\; P \; + \; \mathbf{1}. \bullet (P)$$

Here, P can either execute, or wait. As can be seen in this recursive definition, an asynchronous process may very well stall forever, although this is statistically unlikely.

The strict sequential execution of musical events is often required. To play sequentially, one musician's musical part should end before the next musician commences. A sequential composition operator ";" is useful for this purpose:

$$X \; ; \; Y \;\; = d = \;\; (X[c/\surd] \; \# \;\; \bullet \, (-c.Y)) \setminus \{c\}$$

where c is a new particle not generated by X or Y. The ; operator takes two processes X and Y as arguments, where X must be well-behaved. Here, X will play until it is done, at which time it generates the termination signal $\surd$. This is renamed to c when it is seen, and Y has this same c prefixed to it. When used with the restriction of c, the expression disallows c from appearing. Fortunately, when $\overline{c}$ is finally generated by the left-side, it synchronizes with c on the right, and that particle disappears, ie. $(-c, c) = \mathbf{1}$. Then Y may proceed to play. Note that we do not want X to literally die (reduce to $\mathbf{0}$), or else the whole expression will quit, since $\mathbf{0} \# E = \mathbf{0}$. Rather, X should be well-behaved, remaining silently active in order for the whole composite expression to execute to completion.

3.4 Other implementation issues

MWSCCS has been implemented in MacProlog 32 and SICSTUS Prolog under IRIX 5.3 Unix. The implementation uses a meta-interpreter over MWSCCS expressions. The implementation of various operators were directly derivable from the transitional inference semantics of the original algebra. The

utility of this formal semantic specification for the language cannot be overstated, as it permitted the the quick derivation of a correct and functional implementation.

However, the denotational semantics of concurrent composition # in the process algebra is defined over an exhaustively complete universe of behaviour. This definition is not suitable for use in the implementation, since exponential space and time is required for its construction. Therefore, concurrent composition is implemented with depth-first inferential search. The advantage of this is the avoidance of exponential resource usage; the possible disadvantage is inefficient run-time execution search should an MWSCCS program be ill-structured. This implies that MWSCCS programs should be structured for efficient execution in mind, rather than rely on the execution engine to find results. The use of search is, of course, naturally supported by Prolog.

4 Example

4.1 Grammatical composition structure

Using the notion of well-behaved termination, WSCCS can duplicate the expressiveness of regular grammars, and therefore permits grammatical description of composition (Roads, 1979) For example, we might like to structure a composition as:

$$
\begin{array}{rcl}
Tune & =d= & Prelude\ ;\ Main\ ;\ Finale \\
Prelude & =d= & 2ccdce.\surd.Silent \ + \ cdcee.\surd.Silent \\
Main & =d= & FirstPart\ ;\ SecondPart\ ;\ Climax\ ;\ Resolution \\
FirstPart & =d= & 2ccdce.FirstPart \ + \ cdfee.FirstPart \ + \ g.\surd.Silent \\
\ldots & & \\
Finale & =d= & d.d.b.d.ceg.\surd
\end{array}
$$

Here, there are one of two possible preludes possible, followed by the main body and finale. *FirstPart* has the regular expression form $(X + Y)^* Z$.

4.2 A more complex example

$$
\begin{array}{rcl}
Music & =d= & ((Melody\ \ [x/e5, x\dagger/e5\dagger, b/\surd] \\
 & & \ \#\ \ Acc\_7\_9(x, e5, b)\ \backslash\{\alpha, x, x\dagger, b\})\ [y/ef5, (y\dagger)/(ef5\dagger), b/\surd] \\
 & & \ \#\ \ Acc\_7\_9(y, cf5, b))\ \backslash\ \{\alpha, y, y\dagger, b\} \\
Melody & =d= & (M1\ ;\ M2)\ ;\ (M2\ ;\ M1)\ ;\ \mathbf{0} \\
M1 & =d= & (2,1)\Delta(c5, e5, g5).\mathbf{1}.(c5\dagger, e5\dagger, g5\dagger, \surd).Silent \\
 & & + (1,1)\Delta(cf5, ef5, g5).\mathbf{1}.(cf5\dagger, ef5\dagger, g5\dagger, \surd).Silent \\
M2 & =d= & (2,1)\Delta(c5, ef5, g5).\mathbf{1}.\mathbf{1}.(c5\dagger, ef5\dagger, g5\dagger, \surd).Silent \\
 & & + (1,1)\Delta(c5, ef5, gs5).(c5\dagger, ef5\dagger, gs5\dagger, \surd).Silent \\
Acc\_7\_9(\alpha, \beta, \pi) & =d= & (1,2)\Delta(-\pi, \surd).Silent \\
 & & + (3,1)\Delta(-\alpha, \beta, f(\beta + 7)).Rel\_7\_9(R, X, 7, \pi) \\
 & & + (2,1)\Delta(-\alpha, \beta, f(\beta - 9)).Rel\_7\_9(R, X, -9, \pi) \\
 & & + (1,0)\Delta\mathbf{1}.Acc\_7\_9(\alpha, \beta, \pi) \\
Rel\_7\_9(\alpha, \beta, \gamma, \pi) & =d= & (1,2)\Delta(-\pi, -\alpha\dagger, \beta\dagger, f(\beta + \gamma)\dagger, \surd).Silent \\
 & & + (1,1)\Delta(-\alpha\dagger, \beta\dagger, f(\beta + \gamma)).Acc\_7\_9(\alpha, \beta, \pi) \\
 & & + (1,0)\Delta\mathbf{1}.Rel\_7\_9(\alpha, \beta, \gamma, \pi)
\end{array}
$$

This example uses both grammatical structure, stochastic nondeterminism, and reactive communication. It also uses the *Note On* and *Note Off* notation suitable for MIDI generation. The main

musical process is $Music$, which plays a basic melodic line (*Melody*) along with two accompaniment processes (*Acc_7_9*). Melody uses two subprocesses *M1* and *M2*, which stochastically generate simple chords. *Music* then uses particle renaming when invoking *Acc_7_9*, and restricts the renamed particles so that they will be invisible to the audience. This style of relabelling of events and restricting their emmission is a technique used by process algebras to control communication. The calls to *Acc_7_9* make generous use of argument passing, to make that process as general as possible.

Acc_7_9 does three things, which are prioritized as follows. Firstly, the $(1,2)\Delta$ term checks if *Melody* has terminated, and if so, it quits as well. Otherwise, it checks if *Melody* has generated the event α. If so, it chooses to generate either a note 7 semitones higher, or a note 9 semitones lower. The process *Rel_7_9* is used to release the accompanied tone previously generated. Finally, if neither of the above cases have occurred, *Acc_7_9* waits until the next communicated event.

The use of priorities in *Acc_7_9* and *Rel_7_9* serves two purposes. Firstly, they allow the processes to behave correctly. For example, if we used the same priority value in all the terms of *Acc_7_9*, it is possible that the silent waiting term might be selected when in fact *Melody* has terminated, which prevents the generation of the note off and termination signal actions. Secondly, from an efficiency point of view, priorities greatly reduce the need for search. The most critical actions to take have higher priority and are tried first.

5 Conclusion

Work is under way in using MWSCCS for serious music compositions. One composition being undertaken uses MWSCCS to simulate self-organizing behaviour. In particular, Tofts' WSCCS simulations of ant colony behaviours in (Tofts, 1992) have interesting applications in music. One such MWSCCS composition exploits autosynchronizing behavior which, in a stochastic setting, generates interesting cyclic musical activity. MWSCCS will also be used to formally analyze compositions written in it.

A concurrent music language similar to spirit to MWSCCS is the Petri net (PN) language in (Haus & Sametti, 1992). The similarities and differences between it and MWSCCS are reflected in the inherent differences between process algebraic and Petri net models of concurrency (Nielsen, 1987). Possible advantages of MWSCCS over Petri nets are its more abstract view of musical behaviour, it's lean but powerful set of operators, and its intuitive execution semantics. It is also worth mentioning the similarity between MWSCCS and other conventional music programming languages (Loy & Abbott, 1985), especially those using object-orientation. The main advantage shared by WSCCS and the PN language in comparison to other music languages are their well-defined mathematical foundation, which permits direct formal analyses of systems built with them.

Acknowledgement: Support through NSERC Operating Grant 0138467 is gratefully acknowledged.

References

Aalbersberg, I.K., & Rozenberg, G. 1988. Theory of Traces. *Theoretical Computer Science*, **60**, 1–82.

Chemillier, M. 1990. Solfege, Commutation Partielle et Automates de Contrepoint. *Math. Inf. Sci. Hum.*, **28**(110), 5–25.

Chemillier, M., & Timis, D. 1988. Toward a theory of formal musical languages. *Pages 175–183 of: ICMC 95.*

Haus, G., & Rodriguez, A. 1988. Music Description and Processing by Petri Nets. *Pages 175–199 of:* Rozenberg, G. (ed), *Advances in Petri Nets (LNCS 340).* Springer-Verlag.

Haus, G., & Sametti, A. 1992. ScoreSynth: A System for the Synthesis of Music Scores Based on Petri Nets and a Music Algebra. *Pages 53–77 of:* Baggi, D. (ed), *Computer-Generated Music.* IEEE Computer Society Press.

Hennessy, M. 1988. *Algebraic Theory of Processes.* MIT Press.

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Prentice–Hall.

Loy, G., & Abbott, C. 1985. Programming Languages for Computer Music Synthesis, Performance, and Composition. *Computing Surveys*, **17**(2), 235–265.

Milner, R. 1989. *Communication and Concurrency.* Prentice Hall.

Nielsen, M. 1987. CCS - and its Relationship to Net Theory. *Pages 393–415 of:* Brauer, W. (ed), *Petri Nets: Application and Relationship to Other Models of Concurrency (LNCS 255).* Springer-Verlag.

Peterson, J.L. 1977. Petri Nets. *Computing Surveys*, **9**(3).

Roads, C. 1979. Grammars as Representations for Music. *Computer Music Journal*, **3**(1), 48–55.

Ross, B.J. 1995a (February). *A Process Algebra for Stochastic Music Composition.* Tech. rept. CS-95-02. Brock University, Dept. of Computer Science.

Ross, B.J. 1995b. A Process Algebra for Stochastic Music Composition. *In: Proc. International Computer Music Conference.*

Tofts, C. 1990. A Synchronous Calculus of Relative Frequency. *In:* Baeten, J.C.M., & J.W.Klop (eds), *CONCUR 90.* Amsterdam, The Netherlands: Springer-Verlag LNCS 458.

Tofts, C. 1992. Describing Social Insect Behavior Using Process Algebra. *Transactions of The Society for Computer Simulation*, **9**(4), 227–283.