Arno: Constraints Programming in Common Music

Torsten Anders
Studio für elektroakustische Musik (SeaM)
Hochschule für Musik Franz Liszt
Weimar, Germany
torsten.anders@hfm.uni-weimar.de

Abstract

Constraints programming allows the composer to synthesize a score by describing it. Arno is a program for computer assisted composition which extends Common Music (CM) by means of constraints programming using Screamer. In Arno parameters of CM elements in a CM container can be declared nondeterministically using finite domains — instead of single values. Constraints are expressed as predicates which test one CM object and restrict the actual values of its slots.

Introduction

Music is multidimensional. When listening to music we perceive various aspects such as rhythm, harmony, voice leading or instrumentation simultaneously. During the compositional process as well, the composer intuitively builds up a complex network of relationships between all the musical elements, and observes these elements from various viewpoints more or less simultaneously.

In the domain of computer assisted composition a rich set of compositional strategies exist. Within the paradigm of procedural programming — which is most often employed — the composer has to work and think in a sequential manner, generating data for one particular musical parameter with one function and modifying it with another. Within such procedural programming techniques, however, it is difficult to build up a network of relationships between the various musical elements of a piece. Imagine realizing more than two contrapuntal rules with conditionals like `if` or `case`. In addition, changing or adding a single rule could even require redesigning the entire program.

The paradigm of constraints programming, on the other hand, allows the composer to generate a musical score by describing it. Here, the composer defines the *constraints*, in other words, the properties which the result of the program must fulfil, and the program searches for a solution to satisfy the given constraints. Thus constraints programming frees the composer to concentrate on what he wants to do in a musical sense; the how is left to the computer. Within this paradigm the composer can describe the desired musical result from various viewpoints defining each rule separately.

Constraints programming is already included in other compositional environments such as the PatchWork library *PWConstraints* [Laurson, 1996]. Arno was created to offer a comparatively more flexible constraints environment for the composer.

The Environment Used

Common Music

Arno is an extension to *Common Music* (CM) [Taube, 1994] and is written in *Common Lisp*. It uses the score representation of CM to store its results. In this way, program results can be displayed and edited with the rich CM environment, and the full set of CM output formats is available to the user. Arno, while used with Lisp expressions, is not integrated directly into the *Stella* CM shell.[1]

Screamer

Arno applies *Screamer* [Siskind, 1991; Siskind and McAllester, 1993a,b] in order to implement constraints programming in CM. Screamer complements Common Lisp to make a language efficient in solving numeric and symbolic constraints. Screamer performs a backtracking search.[2]

The fundamental idea of constraints programming is to introduce alternatives — referred to as a *domain* — for a single value and to prohibit specific alternatives through predicates — called *constraints*. The program chooses a possible solution to a problem based on the domains and constraints provided by the user.

To support this programming style Screamer adds nondeterministic generators and the special operator `fail` to Common Lisp. Nondeterministic generators (such as `either`, `a-member-of` and `an-integer-between`) return single values from a set of alternatives. The function `fail` is used to prohibit some situations. In

[1] Common Music is freely available under `ftp://ccrma-ftp.stanford.edu/pub/Lisp/cm/`.

[2] Screamer is freely available under `http://www.cis.upenn.edu/~screamer-tools/home.html`.

Screamer, nondeterministic expressions must be placed in a context which allows their existence (such as carried out by `one-value` or a function definition).

The example in figure 1 declares the variable x with the domain $[1, 2, 3]$. x shall be even; the first possible solution is returned.

```
> (one-value
    (let ((x (either 1 2 3)))
      (unless (evenp x) (fail))
      x))
> 2
```

Figure 1: A nondeterministic expression in Screamer

The Program

Arno provides the composer with a means of programming in constraints. Arno is specialized in the generation of musical scores and is designed to provide as much compositional freedom as possible. In Arno, the definition of *musical form* and *musical constraints* are, in general, separated. Here, musical form refers to how many Common Music objects (i.e. of type `midi-note` or `thread`) are used as well as to object types and to the hierarchical structuring of the containers. Constraints refers to further properties of the elements within the containers.

In Arno the form is defined by the macro `defcontain`. This macro initializes the desired content of a CM container and initiates the search. It understands some keyword arguments like `object-type` and `number` of the content, as well as how to set parameters of the content and a list of constraints (`fulfil` or `avoid`). `defcontain` defines a named function with at least one argument, the function expects the container to be filled as declared in the body of `defcontain`.

The description of the form can be hierarchically structured: containers may contain containers. To obtain this, the content type of a `defcontain` expression must be a container instead an element, and the function defined by another `defcontain` expression must be called in the content setting. This allows the expression of polyphony (CM containers of type `thread` in a `merge`).

Constraints are defined one by one as predicates which expect a single argument — the current element to be tested. In this way single properties of the result can be described separately. A constraint is used by including its name in a list which is given to `defcontain` as an argument. Constraints can be independently added to, removed from or exchanged within the list.

In the paradigm of constraints programming, the program has to choose a value — allowed by the constraints — from a set of possible values. Thus this set of possible values, referred to as the domain, must first be declared. In Arno nondeterministic generators available through Screamer are used to return not simply a value but rather a nondeterministic value, i.e. one of a set of possible values. Which value is finally used will depend on further computations, i.e. the applied set of constraints.

The domain for each parameter of the elements in a container is declared in `defcontain`. Any expression which returns a nondeterministic value can be used.[3] Every parameter, including the rhythm, duration, note or any synthesis parameter, can be declared in this way. Thus the composer is free to declare domains of microtonal pitches using floats or ratios for the frequency. Rule based rhythmic structures are also possible in conjunction with rhythm-related constraints. The domain for the parameter of an element can be dependent of its predecessors — this can be useful for realizing heuristics.

In the defining constraints, any relationship between the various parameters of any notes can be declared. Arno uses the CM score representation to store its preliminary results during search. Because every CM element "knows" its container, one can address elements in other positions, such as previous elements (elements in the same container) or simultaneous elements (elements in the same merge at the same time). Elements are addressed using functions of the CM API. In Arno, this set of API functions is extended with functions such as `previous-objects` and `simultaneous-objects`. Thus using constraints the composer can declare voice leading rules involving the pitch of the current and some previous elements. The resulting harmony can be controlled by defining constraints for the pitches of simultaneous elements.

Arno introduces the concept of time dependent domains and constraints with envelopes. An envelope — which embraces a container — returns a position-dependent value for each element. These values can freely be used in the definition of the constraint or the domain declaration.

Because CM and Screamer are highly portable and because Arno uses no platform specific code, Arno itself is portable as well. Its portability was successfully tested under *Allegro Common Lisp 4.3* on Linux and *Macintosh Common Lisp 4.0*. The Arno source is freely available.

Example 1: An All-Interval Series

As an example, a declaration of an all-interval series is presented. Figure 2 shows the definition of the function `all-interval-series`. This function expects a CM container and fills it with 12 `midi-notes`. The domain

[3] Of course it is also possible to declare parameters purely deterministically.

for the `note` slot of each `midi-note` consists of the integers 60–71 in a shuffled order. Two constraints must be avoided to assure an all-interval series. The function `all-interval-series` iterates over all elements in its container. The macro `current-object` returns the current element during the loop.

```
(defcontain all-interval-series
    :content-type (object midi-note rhythm 1)
    :number 12
    :content-setting
    (set-object (current-object)
                'note (a-shuffled-expr
                       (an-integer-between 60 71)))
    :avoid '(duplicate-note? duplicate-interval?))
```

Figure 2: Definition of the form for an all-interval series

In figure 3 the two constraints are defined as predicates with one argument. `duplicate-note?` tests whether the note name of its argument is also the note name of a predecessor. It uses two Arno functions: `get-note` is simply a slot accessor for a note object; `previous-objects` returns all objects previous to its argument in the same thread in backward order (nearest object first).

```
(defmethod duplicate-note? ((note midi-note))
  (let ((prev-notes
         (mapcar #'(lambda (n) (get-note n))
                 (previous-objects note))))
    (member (get-note note) prev-notes)))

(defmethod duplicate-interval? ((note midi-note))
  (let*-when ((prev-obj
               (previous-object note))
              (prev-notes
               (mapcar #'(lambda (n) (get-note n))
                       (reverse
                        (previous-objects note))))
              (prev-intervalls
               (mapcar #'(lambda (pre succ)
                           (upward-interval pre succ))
                       (butlast prev-notes)
                       (rest prev-notes))))
      (member (upward-interval  (get-note prev-obj)
                                (get-note note))
              prev-intervalls)))

(defun upward-interval (pre succ)
  (let ((int (- succ pre)))
    (if (< int 0) (+ int 12) int)))
```

Figure 3: Definition of two constraints for an all-interval series

The predicate `duplicate-interval?` assures that the interval between its argument and the predecessor of this argument is unique. Complementary intervals in opposite directions are treated as the same interval. Therefore, the auxiliary function `upward-interval` calculates the interval between two notes, but downward intervals are converted to their complementary upward counterpart.

The Arno macro `let*-when` is very similar to the Lisp primitive `let*`, but tests every variable-binding to be non `NIL`. Otherwise, the whole expression returns `NIL`. The function `previous-object` returns the very predecessor of an object. The order of the previous-objects is reversed in order to place the first object of a container at the initial position.

The example is evaluated by calling the function `all-interval-series` with a CM `thread`. Because `all-interval-series` is a nondeterministic function it must be called within the Screamer macro `one-value`.

```
Stella [Top-Level]: (thread all-interval-series ())
#<THREAD: All-Interval-Series>
Stella [Top-Level]: (one-value
                     (all-interval-series
                      #!all-interval-series))
#<THREAD: All-Interval-Series>
```

Example 2: A Canon

The next example is a two-voice canon with voice leading and harmonic constraints. The example is kept simple in order to demonstrate the underlying principle. The form is defined in the figure 4. The definition of `first-voice` is similar to the definition of `all-interval-series`. In the function `other-voice`, the slots of midi-notes of the first-voice are simply copied. Two auxiliary functions are defined for this purpose in figure 6. Both functions are combined by the definition of the function `canon`.

In defining the two contrapuntal voices, the rhythm of the midi-notes is initialized with 0 — the rhythm slot must contain a numeric value.[4] The frequently occurring acronym BJ in the definitions stands for *backjumping*, a search strategy similar but more efficient than *backtracking* which is the default strategy.[5]

Two constraints for the canon are defined in figure 5. `not-allowed-jump?` ensures that an interval between two neighboring notes in the same thread is a fifth or less. `dissonant?` tests the interval between simultaneous notes in different voices. The interval between simultaneous notes must be a minor or major third, a fifth or a minor or major sixth.[6]

[4] Arno updates the time slot of every note before each new search step. This updates the time-dependent relationships between the notes (which is noted by functions like `simultaneous-objects`) and is the reason why the rhythm slot must always be initialized.

[5] Arno binds the slots of all elements in one container before going on to the next. This means that discrepancies between simultaneous elements (in different containers) will be found rather late. The backjumping search strategy is more efficient than chronological backtracking because it jumps directly back to a conflicting element as soon as it finds a discrepancy.

[6] If the constraint fails it returns the first simultaneous note as the target for backjumping.

```
(defparameter *note-number* 9)

(defcontain first-voice
    :content-type (object midi-note rhythm 0)
    :number *note-number*
    :content-setting
    (set-object (current-object)
                'rhythm (a-shuffled-expr-bj
                         (either 2 1))
                'note (a-shuffled-expr-bj
                       (either 60 62 64 65 67 69 71 72)))
    :avoid '(not-allowed-jump?)
    :bj? T)

(defcontain other-voice
    :content-type (object midi-note rhythm 0)
    :number *note-number*
    :content-setting
    (copy-slots (matching-note (current-object))
                (current-object))
    :avoid '(dissonant?)
    :bj? T)

(defcontain canon
    :content-type (make-object 'thread)
    :number 2
    :content-setting
    (let ((c (current-object)))
      (case (object-position c)
        (0 (first-voice c))
        (1 (set-object c 'start 3)
           (other-voice c)))))
```

Figure 4: Definition of the form of a two-voice canon

Further Development

Arno is currently a working prototype and is under modification to improve its performance. The search algorithm, for example, uses a static variable order. This causes a certain amount of redundant work.[7] A dynamic variable order which addresses this problem is under development.[8] The Lisp implementation of the most often visited parts of the code must also be optimized.

Various extensions are projected. The idea of time dependent constraints will be further developed with the idea of using not only envelopes but CM item streams in the declaration of domains and the definition of constraints. Long term goals involve the use of weighted constraints — which could be classified by degree of importance — and the possibility of varying the content of a container once it is built up instead of constructing it from scratch every time.

[7]Since the program generates the rhythmic structure of polyphonic music too, it cannot guess in advance, which events will be simultaneous in time. Therefore there can be no static variable order in order to avoid redundant work.

[8]The algorithm must know the time structure of the temporary result and, in proceeding, must jump between different containers.

```
(defmethod not-allowed-jump? ((note midi-note))
  (let*-when ((prev (previous-note note))
              (int (- (get-note note)
                      (get-note prev))))
      (not
       (< 0 (abs int) 8))))

(defmethod dissonant? ((note midi-note))
  (let*-when ((sims (remove-if
                     #'(lambda (n)
                         (= (get-rhythm n) 0))
                     (simultaneous-objects note)))
              (ints (mapcar #'(lambda (n)
                                (- (get-note note)
                                   (get-note n)))
                            sims)))
      (when
          (find-if-not #'(lambda (int)
                           (member (mod (abs int) 12)
                                   '(3 4 7 8 9)))
                       ints)
        (first sims))))
```

Figure 5: Definition of some constaints for the canon

```
(defmethod matching-note ((note element))
  (nth-object (object-position note)
              (nth-object
               0
               (object-container
                (object-container note)))))

(defmethod copy-slots ((note-orig midi-note)
                       (note-copy midi-note))
  (set-object note-copy
              'rhythm (get-rhythm note-orig)
              'note (get-note note-orig)))
```

Figure 6: Auxiliary definitions for the canon

References

Mikael Laurson. *PatchWork, PWConstraints*. IRCAM, Paris, first english edition, Oktober 1996. Reference manual.

Jeffrey Mark Siskind. *Screaming Yellow Zonkers*. MIT Artificial Intelligence Laboratory, 1991.

Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic Lisp as a Substrate for Constraints Logic Programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993a.

Jeffrey Mark Siskind and David Allen McAllester. Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp. Technical report, University of Pennsylvania Insitute for Research in Cognitive Science, 1993b.

Heinrich Taube. Stella: Persistent Score Editing in Common Music. *Computer Music Journal*, 17:4, 1994.