Mapping with planning agents in the *Max/MSP* environment: the *GO/Max* language

Paolo Bottoni, Stefano Faralli, Anna Labella, Mario Pierro
Università di Roma La Sapienza Dipartimento di Informatica
Via Salaria, 113, 00198 – Rome, Italy
(bottoni,faralli,labella, pierro)@di.uniroma1.it

ABSTRACT

GO/Max is an agent programming language that facilitates the design of algorithms for real-time control of sound/music generation programs crafted in the Max/MSP environment. We show how software planning agents programmed in GO/Max can be used to transform abstract goal states specified by the performer in potentially complex sequences of Max/MSP control messages.

Keywords

mapping, planning, agent, Max/MSP

1. INTRODUCTION

In the context of real-time interactive musical systems, *mapping* represents the correspondence between the control space in which the performer acts and the parameter space of the system that is being controlled. The central role of mapping and the importance of having efficient methods to design mappings have been exposed by several works [1].

Several approaches for translating from the control space into the parameter space have been proposed (see section 1.1). In this paper we present our experiments on using *state models* and *planning agents* to perform such translation. The basic idea behind this approach is to represent the controlled system using a *state model*. If a state model for the system is finite, it is possible to implement a software agent that will be able to foresee the evolution of such model using a *planning* algorithm [2]: the agent will be able to produce a command sequence that leads from one model state to another goal state, provided that such transition is possible, or state its non-existence. If the model also specifies a correspondence between state transitions in the model and commands for the system, a sequence of states in the model can be used to drive the actual system.

The combination of the system model and the planning agent will thus form an *abstraction* layer between the performer and the controlled system. The performer will drive the agent by sending it requests for *goal* states; the agent will devise sequences of operators to reach those goal states; operator sequences correspond to message sequences used to steer the abstracted system (see Figure 1).

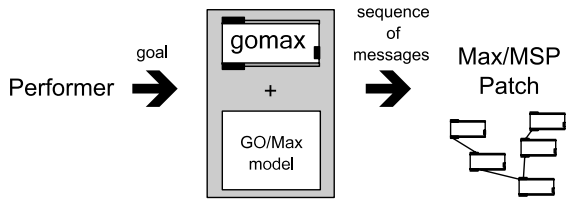

Figure 1: Performer requests a goal to the gomax agent, agent's plans to reach the goal are translated in control sequences for the Max/MSP patch.

The abstraction is obtained because of the independence between goal/operator definitions and messages which are sent to the patch. This can be particularly effective when used in conjunction with other mapping techniques which can be used to form goal state requests depending on the performers input. One example could be the mapping of a discrete set of hand gestures to a set of agent goal states.

We have experimented abstracting the control of Max/MSP patches. To do so we developed a language (*GO/Max*, standing for Goal-Oriented Max) to describe deterministic state models of Max/MSP patches, and a Max/MSP external (`gomax`) to implement a software agent which performs planning and translates state model transitions into actual patch commands [3].

1.1 Related Work

The mapping problem has been approached in numerous ways: [4] is a comprehensive review of mapping techniques. Generally mapping is a *one-to-one*, *one-to-many* or *many-to-one* correspondence between the control space and the parameter space, where their dimension numbers generally differ. Van Nort et al. [5] have formalized mapping as a continuous function from the control to the parameter space and, representing those as subsets of a highly-dimensional Euclidean space, have analyzed the geometrical and analytical properties of that function.

Some continuous mapping algorithms such as Escher [6] use an intermediate layer of abstract parameters to translate between the control space and parameter space. The recent MnM toolbox [7] uses matrices to represent mappings and provides methods to build them iteratively. Many-to-one mappings have also been approached as a *pattern recognition* problem to which neural networks have been applied (most recently in [8]). The recent proposal of `ChucK` ([9], [10]) has exposed the central role played by the programming paradigm in the context of high-level com-

NIME 06, June 4-8, 2006, Paris, France

puter music languages, and specifically in the definition of complex mappings.

2. PLANNING AND MAPPING

The task of *planning* is generally summarized as "finding a sequence of actions that will achieve a goal"[11].

A *deterministic state model* is defined as a set of states S, a set of actions A, and a transition function $f : S \times A \rightarrow S$ which describes how actions map one state into another. If the system to which actions are applied is represented by a deterministic state model, the planning problem is a *deterministic control problem* [2].

An instance of a planning problem can be described using a formal language: *STRIPS* [12] is the classic example on which also GO/Max is based. A STRIPS description of a planning problem describes a deterministic state model in which the action costs are all equal [2]. If the agent environment is "fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects)" then the planning task is called *classical planning* [11].

A planning problem instance described in GO/Max is a tuple $P = \langle L, A, I, G \rangle$ where:

L is a set of literals (*atoms*), so that every state s of the state model is a subset of L: $\forall s \in S, s \subseteq L$. In a GO/Max program, literals can be defined explicitly

```
literal someBooleanCondition;
```

or they can be implicitly specified using variables on finite domains. Domains can be either integer intervals or explicit lists of strings:

```
domain intDomain: 0..8;
domain listDomain: {a, b, c, d};
var someVariable: intDomain;
```

A is a set of *actions*. For every $a \in A$ we define the *preconditions* set $pre(a) \subseteq L$. An action is applicable in a state s if $pre(a) \subseteq s$; the *postconditions* describe the effects of the action and are represented by a set pair $post(a) = (add(a), del(a))$ where $add(a) \subseteq L$ and $del(a) \subseteq L$. The state s' resulting from the execution of the action a in a state s is defined as $s' = (s \cup del(a)) \backslash add(a)$. Extending the syntax of the traditional STRIPS language, every operator is described in terms of:

- *parameters*, as local GO/Max variables,
- *preconditions*, either expressed as literals or as boolean conditions on the variables,
- *postconditions*, either expressed as literals, negated literals (using the `!` operator) or as GO/Max variable assignments (using the `:=` operator)
- a Max *message*, which will be sent during the operators execution. Message syntax is a subset of the one used in the standard Max *message box* object, so a message can be composed of integers, floats, bang and symbols. Several messages can be sent sequentially by separating them with a comma. It is not possible to use the semicolon to send the message to a specified receiver object. An expression evaluation operator `&( )` has been added to create integer and symbol values from GO/Max variables with integer and string-list domains respectively, as well as expressions containing integer variables.

Here is an example of an operator declaration:

```
operator someOperator( someParameter: intDomain )
pre( someBooleanCondition, someVariable < 3 )
out( set &(someVariable) )
post( someVariable := someParameter );
```

This operator will allow the agent to change the value of `someVariable` to any of the values in `intDomain`, provided that the `someBooleanCondition` literal is present in the current state, *and* that the current value of `someVariable` is less than 3. Every time an agent will use this operator, it will choose a target value in `intDomain`, output the Max message `set` followed by the current value of the variable `someVariable`, and finally update the variable's value to the chosen target value.

$I \subseteq L$ is the initial state, declared explicitly using the `state` construct:

```
state(someBooleanCondition, someVariable=0)
```

In this case, the literal `someBooleanCondition` is included in the start state, and the initial value of `someVariable` is zero.

G is the goal description, in terms of a set pair (G_{pos}, G_{neg}) where $G_{pos} \subseteq L$ and $G_{neg} \subseteq L$: a state s is a goal state if $G_{pos} \subseteq s$ and $G_{neg} \cap s = \emptyset$. Goal states are specified at run-time, sending the `goal` message to an agent (see 2.1). For example:

```
goal someBooleanCondition someVariable=100
```

Every literal in a GO/Max model corresponds to a boolean condition in the agent environment. A *closed world assumption* is adopted, so that if a state does *not* contain some literal l, then the boolean condition associated to l is *assumed* to be false. Every state is then a complete description of the agent environment.

Given a STRIPS-like problem description, a *planning agent* autonomously finds an action sequence leading from the initial state to the goal state, if such sequence exists. A problem description in a suitable language, such as GO/Max , can therefore be used as a declarative-paradigm agent programming language. This allows the programmer to specify just *what* should be done instead of *how* it should be done, and assures that all states reached during program execution respect the model declaration. By declaring the operations and the initial state, a GO/Max program potentially describes different execution sequences for them, thus being equivalent to several iterative programs. Compared to existing procedural-code features of Max (Javascript, Java, RTCmix), this adds compactness and flexibility to programs, especially in mapping applications where the agents can automatically devise a new action sequence if a new goal request occurs.

States in the model can be mapped to configurations of parameters in continuous data spaces, so that other interpolation techniques can be used to translate from one configuration to another.

Note that since the actions are *assumed* to have deterministic effects, the agent can operate without reading any feedback from the actual state of the system (Fig. 1). Also, classical planning relies on *atomic* plan generation and execution: the environment does not change while the agent is calculating the plan (see 2.2).

The classical planning problem is a well-known AI topic and has been approached with many different algorithms ([11], [13], [14], [15]). STRIPS planning is a highly complex computational task [16] . Nevertheless, planning is a

very powerful tool even for models with small state spaces, and is used in many applications which include robotics and computer games.

2.1 The *gomax* software agent

The GO/Max patch model can be compiled by a `gomax` software agent which operates in the patch itself as an external object. A model is compiled with the message `parse (filename)`.

A performer interacting with the patch will be able to issue requests for goal states to the agent using the `goal` message. A goal state will be described in terms of literals in the currently compiled model. When a request for a goal state is made, since every operator is associated with a Max message, the computed sequence of operators is associated with a sequence of Max messages that are sent out via the agent's outlet. While the goal state is specified by the performer, the way in which it is reached will be determined by the agent based on the GO/Max model that has been compiled and on the search algorithm used.

2.2 Patches and state models

As we can see from the scheme in Figure 1, `gomax` agents currently do not receive any feedback from the patch, so it is *assumed* that the actual state of the patch reflects the state of the GO/Max model which is used for planning. This obviously limits the structure of patches which can be represented by a GO/Max model, and it is now needed because the algorithms used to implement planning in the agent do not allow for a non-deterministic effect of actions. Building a model whose changes reflect the patch changes is the model designer's responsibility. Also, since classical planning is used, the modeled patch is expected to change only when the agent acts.

Max/MSP has a well-defined semantics for its patches, where the order of activation of the connections between objects is based on the objects positions and right-to-left inlet ordering [17].

If a patch contains a purposely non-deterministic section, such as a random number generator, that section can be either ignored in the GO/Max model or used to generate goal requests for the agent so that the internal state of the model can be re-aligned with the actual state of the patch.

2.3 Planning algorithms and real-time performance

Since there is no optimal algorithm for planning [11], it is possible to choose which algorithm the `gomax` agent should use by means of the `search` message. This will also determine the behavior of the agent when multiple equivalent plans are found: while normal algorithms will account for operator declaration order to choose which plan will be used, the provided "non-deterministic" variants will choose a plan at random.

In the present version the available algorithms are a simple breadth-first search and a heuristically-guided A-star search. Both operate in the state space and are provided in deterministic and non-deterministic variants. The A-star search is a modified version of the algorithm described in [13], the details of the algorithm are described in [3]. More search algorithms will be added as new versions of the agent will be released.

The current algorithms allow real-time use of the agent with models containing up to about 200 states, even if this greatly depends on the structure of the model. With a model of this size, plan generation time is under 0.1 seconds using an Intel Pentium 4 1.8Ghz processor. It is expected to optimize plan generation time further in the next agent versions.

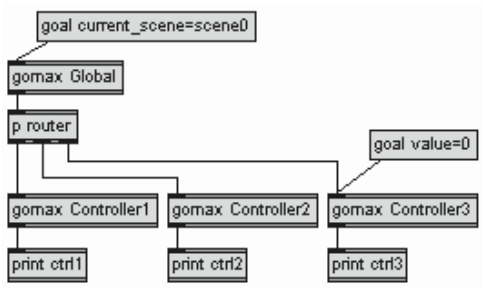

Figure 2: Using multiple agents

However, even with very simple models, using an abstraction layer is a fast way to implement mappings which would be difficult to realize with other techniques. Subdividing a large model into independent sub-models can reduce its computational cost (see 2.5).

2.4 Models and time representation

Once a plan is determined, a gomax agent fully outputs the corresponding sequence of messages *atomically*. The resulting messages can then be, for example, stored in a buffer and read every n milliseconds, so that no explicit notion of time is needed in the model. A different model could output messages with a specific timestamp, thus making the buffer a sequencer. The GO/Max language does not offer any *predefined* construct to specify an operator execution order. Time can be represented in the model used, so that, for instance, certain precedences between operators are established ("always choose a note length before playing a note") or that messages output by the agents carry a timestamp, which is the case when the "abstracted" patch has a sequencer-like behavior.

2.5 Multiple agents

Max messages associated with GO/Max operators are arbitrary and can be used to issue `goal` messages to other `gomax` agents, thus decomposing a large model in smaller independent models, each handled by a separate agent.

The different agents coordinate their operations using the Max/MSP depth-first message handling scheme [17]. A plan generation started from an agent will not return control to the patch until all sub-agents have either generated a plan or returned an error. Also, the plan-starting agent will not update its internal state nor continue the execution of its plan if, at some point during plan execution, one of the sub-agents has reported an error.

In the example of Figure 2, a system containing three independent parameters has been modeled using three separate GO/Max models/agents, and a "master" agent which sends individual goal requests to each. If the parameters are independent, this distributed model offers the same functionality as a single model containing all three parameters, but uses a smaller state space. For example, if each of the three parameters has a 0-127 range, a single model will contain 127^3 states while a distributed model will have only $127*3$. This kind of model decomposition is often a feasible solution to allow real-time control of large patches. Using different agents, every agent offers a different abstraction layer to the performer. Since agent op-

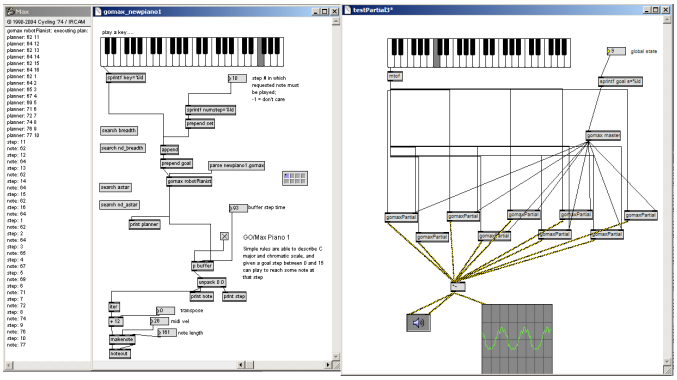

Figure 3: Meta-Piano and Additive patches

erations are always coordinated, the performer can freely switch between the different abstraction layers to control the system in different ways.

3. EXAMPLES

Since GO/Max models use standard Max messages, they have a broad range of applications (music, graphics, etc.).

3.1 Meta-Piano

The model used in this patch represents a 16-step buffer and a writing position inside it. Operators output a buffer step value (`key` variable) and its position (`numstep`), rolling back to zero when the last step is reached. The `key` variable value is a MIDI note pitch, and the operators change its value following rules such as chromatic/major/minor scales, predefined note successions, etc.. When the performer chooses a target note pitch and a target buffer position (Fig. 3) the agent will find a succession of notes that follows the rules specified by the operators and ends exactly on the required step. An example operator is:

```
var key: 36..83; var numstep: 0..15;
operator SemitoneUp
    pre ( )
    out ( &(key + 1) &(numstep + 1) )
    post( key := key + 1,
          numstep := (numstep + 1) % 16 );
```

3.2 Additive Synthesis

Here different agents control partials independently, each using a different model, constructed so that the operators change the agent's state and output fixed values for the partial's frequency, phase and amplitude (Fig. 3). A buffer is used to output agent messages at a fixed rate, and *line~* objects smooth transitions between values. A further "master" agent can send goal requests to all subagents as shown before (see 2.5). When the performer asks one partial or master-agent to reach a goal state, the activated agent will determine a sequence of its model operators which will respectively correspond to a sequence of partial parameters, or to a sequence of goal requests for all the partials simultaneously. Control is therefore abstracted by generating goal requests from an input parameter, indicating the target sound state. Here we show a short model for one of the partial-agents, which outputs constant phase and volume values and links the partial's frequency to the model's state number, establishing a simple state transition scheme:

```
var pState: 0..10;
operator SwitchToNextState
pre ()
out ( &(pState*50) 0.0 1.0 )
post( pState := (pState + 3)%10 );
state ( pState=1 )
```

When a target value for `pState` is specified, the partial's frequency will thus change in a model-dependent way.

4. FUTURE WORK AND CONCLUSIONS

GO/Max brings the benefits of a high-level, declarative language to Max/MSP, and is currently in beta stage for its 1.0 release. On the language side, an interesting development would be a visual editor to construct a specific subset of GO/Max models. This could be an evolution of the traditional one-to-one *MIDI Learn* facility. On the agent side, we plan to work on an explicit feedback mechanism and on asynchronous (non-blocking) plan generation. We also plan to port GO/Max to other computer music languages such as ChucK or OSC. To achieve this, a suitable state model must be determined as well as efficient planning algorithms for that model.

5. REFERENCES

[1] A. Hunt M. Wanderley and M. Paradis. The importance of parameter mapping in electronic instrument design. In *Proc. of NIME-02*, pages 149–154, 2002.

[2] B. Bonet and H. Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

[3] M. Pierro. Sistemi di controllo per la generazione musicale, Tesi di Laurea, Corso di Laurea in Informatica, University of Rome "La Sapienza", 2005. *http://hci.uniroma1.it/multimedialab/*

[4] M. Wanderley and M. Battier, editors. *Trends in Gestural Control of Music.* IRCAM, 2000.

[5] D. Van Nort, M. Wanderley and P. Depalle. On the choice of mappings based on geometric properties. In *Proc. of NIME-04*, pages 87–91, 2004.

[6] M. M. Wanderley, N. Schnell, and J. Rovan. ESCHER—Modeling and Performing composed Instruments in real-time. In *IEEE Systems, Man, and Cybernetics Conference*, October 1998.

[7] F. Bevilacqua, R. Müller and N. Schnell. MnM: a Max/MSP mapping toolbox. In *Proc. of NIME-05*, pages 85–89, 2005.

[8] A. Cont, T. Coduys and C. Henry. Real-time gesture mapping in pd environment using neural networks. In *Proc. of NIME-04*, pages 39–42, 2004.

[9] G. Wang and P. R. Cook. On-the-fly programming: Using code as an expressive musical instrument. In *Proc. of NIME-04*, pages 138–143, 2004.

[10] G. Wang, A. Misra, A. Kapur, P. R. Cook. Yeah, chuck it! => dynamic, controllable interface mapping. In *Proc. of NIME-05*, pages 196–199, 2005.

[11] S. Russel and P. Norvig. *Artificial Intelligence*. Prentice Hall, 1995.

[12] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proc. of the 2nd IJCAI*, pages 608–620, 1971.

[13] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res. (JAIR)*, 14:253–302, 2001.

[14] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI'92*, pages 359–363, 1992.

[15] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[16] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[17] Cycling '74. *Max Reference manual*, 2004.