IMPLEMENTATION OF A REAL-TIME MUSICAL DECISION-MAKER

Aengus Martin, Craig T. Jin

Computing & Audio Research Lab
Sydney University, NSW 2006, Australia

Oliver Bown

Faculty of Architecture, Design and Planning
Sydney University, NSW 2006, Australia

ABSTRACT

In this paper, we present a novel implementation of a real-time musical decision-maker. In our scenario, the behaviour of a musical decision-making agent is specified as a constraint satisfaction problem. Generally, constraint satisfaction problems are solved using methods which are not suitable for real-time musical performance, because the amount of time they will take to arrive at a solution is unpredictable. We have developed a musical decision-making agent which can solve some musical constraint problems (i) in a predictable length of time and (ii) usually much more quickly than alternative methods. It works by using an efficient data structure called a binary decision diagram to represent the constraint satisfaction problem.

1. INTRODUCTION

We use the term *musical agent* to refer to a computational entity intended to play music either autonomously or alongside a human performer. The computer music literature contains many reports on musical agents (see, e.g. [6] for a review) and a great variety of algorithms and architectures have been used in their implementation [10, 12, 20]. Less attention has been paid to the methodology of designing musical agents—that is, to the approaches to the design of musical agents which are not specific to particular music systems or situations.

We have previously studied a number of approaches to the creation of tools with which musicians can design the musical decision-making behaviour of a musical agent for use in their creative work. We have studied the use of *partially observable Markov decision processes* (POMDPs), whereby a musician can design aspects of an agent's behaviour by tuning the *reward function* of a POMDP model [13]. In addition, we have studied a *design-by-example* approach in which a musician can specify the behaviour of an agent by supplying examples of the desired behaviour [15].

In these studies, the focus was on design methods with two particular characteristics. The first is that they do not require musicians to be experts at computer programming or algorithm implementation or development (though, of course, they may be). The second is that the musician's aim is to design agents which make relatively high-level musical decisions, for example, the sequencing of musical elements such as sound samples and short MIDI sequences; or controlling the parameters of digital audio effects or processes which generate low-level musical material.

Recently we introduced a toolkit for the design of musical agents, which was also created with the aforementioned characteristics in mind [14]. The *Agent Design Toolkit* allows a musician to design the behaviour of a musical agent by first recording some examples of the desired behaviour and then following an iterative design process in which the musician can add to, or edit the examples, as well as adjust the parameters of a set of machine learning algorithms which learn the musical patterns in the example data. At each design iteration, an agent is created, which the musician can audition in order to decide what modifications to make to the examples and/or the parameters of the machine learning algorithms. This paradigm, in which a user interacts with machine learning algorithms iteratively to arrive at a satisfactory result, is known as *interactive machine learning* [7] and it was first used in the context of computer music by Fiebrink et al [8].

The Agent Design Toolkit is comprised primarily of two separate pieces of software. The first is the *Agent Designer*, which is used to record examples and run the machine learning algorithms. Its output is an *agent behaviour specification*; a file which specifies the behaviour of a musical agent. This file is then loaded into the *performer module* for real-time use in a musical performance.

In this paper, we report an a completely new implementation of the performer module, which is much more computationally efficient and is capable of executing far more complex behaviours than was previously possible with the toolkit. It relies on the use of a data structure called a *binary decision diagram* [4], which can be used to solve certain types of computational problems much more efficiently than with alternative methods.

The remainder of this paper is structured as follows. In Section 2, we give an overview of the Agent Design Toolkit and how it is used. In Section 3, we describe the performer module and our new implementation using binary decision diagrams. In Section 4, we discuss the performance improvements using the new performer module. In Section 5, we discuss the Agent Design Toolkit and its application in the context of other research, before concluding in Section 6.

2. OVERVIEW OF THE AGENT DESIGN TOOLKIT

In [14] we describe the Agent Design Toolkit and discuss a preliminary study of its use. In this section, we give a detailed overview of the toolkit. This will provide the context for understanding the issues involved in the implementation of the performer module.

2.1. Design paradigm

Our goal is to enable the creation of musical decision-making agents that can be adapted by music practitioners to their specific creative contexts. In a typical scenario, a musician has assembled a *music system* comprising an ensemble of virtual instruments, digital audio effects and algorithmic processes, all of which can be parametrically controlled (i.e. there is a set of variables whose values determine the output of the music system). An example of such a system is an Ableton Live project (www.ableton.com). In this context, the project is the music system which the musical decision-making agent will control. The Agent Design Toolkit provides a means for a music practitioner who is not an expert in algorithms or computer programming to design the behaviour of such an agent. The agent's decision making process can involve the use of external musical variables, e.g. incoming MIDI data and therefore the Agent Design Toolkit can be used to create both generative and interactive music systems.

2.2. Machine Learning Algorithms

Here, we give an overview of the machine learning algorithms used in the Agent Design Toolkit to learn the musical patterns in the example data. Separate machine learning algorithms are used to model (i) the -dependencies between the variables which control the music system and (ii), the manner in which they change over time. For (i), we use *association rule learning* (ARL) algorithms (see, e.g. [9]). These algorithms can be used to find deterministic rules describing dependencies between the variables. The rules are in the form of *implies* rules. For example, a music system may have separate variables which control the activity of the bass drum, the snare drum and the hi-hat, and the following rule may be discovered:

Bass drum = On, Snare drum = On ➔ Hi-hat = On

which can be read as "when the bass drum and the snare drum are sounding, then the hi-hat is sounding too". The musician can configure various aspects of the rule learning process.

For (ii)—learning the manner in which variables change over time—we use *variable order Markov models* (VMMs) [19]. Given the current value of a variable and its history of previous values, a VMM can be used to efficiently calculate the set of new values that the variable may take next, and their associated probabilities. The musician can choose which variables are modelled by VMMs and the maximum *order* of the VMMs (i.e. the maximum number of historical values to take into account when calculating the probability distribution for the next value that the variable will take). Higher order models produce sequences closer to those found in the set of examples provided by the musician.

As part of the design process, the musician must choose an integer-valued *priority* for each variable with which a VMM is associated. This is necessary for the following reason. At performance time, the agent uses the VMMs to choose values for the variables with which they are associated. As this is done for each variable independently, the VMM-chosen values can be inconsistent with the ARL-derived rules. When a conflict of this kind arises, the VMM-chosen value associated with one or more of the variables must be ignored so that no rules are broken. The agent uses the variables' priority values to choose which ones will retain their VMM-chosen values and which ones will not. Variables with high priorities are more likely to retain their VMM-chosen values in preference over those with low priorities.

2.3. Agent behaviour specification

To summarise, we list the different elements of an agent behaviour specification output by the Agent Designer and used by the performer module:

- **Music system variables:** The attributes of each variable associated with the music system. These include the *domain* (the values which the variable may take); a flag indicating whether the variable is *ordinal* or *categorical* (i.e. whether or not the values in the domain have an intrinsic numerical ordering); a flag indicating whether the variable is *controllable* or not (i.e. whether it is to be controlled by the agent, or just used in the decision-making process); the order of the VMM, if necessary; the *priority* with which the VMM is used; and a flag indicating if the value of the variable should be output at performance time.
- **Custom variables:** The attributes of the custom variables, which are additional, musician-specified descriptors that are intended to help the machine learning algorithms to find salient musical patterns (see [14, 16]).
- **Variable order Markov models:** The VMMs associated with certain variables and custom variables, as chosen by the musician.
- **Rules:** The set of rules describing the dependencies between variables (including custom variables), as discovered by the ARL algorithms.

For the purposes of illustration, Figure 1 shows a screen-shot of the "Variables" panel from the Agent Designer. The panel is used for choosing which variables will have VMMs associated with them, and for defining custom variables.

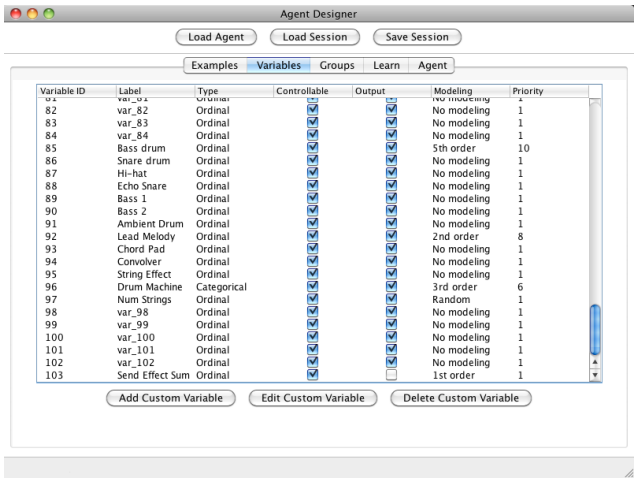

Figure 1. A screenshot of the Agent Designer.

3. THE PERFORMER MODULE

In this section we give details of the performer module. We start by describing the procedure which must be executed by the module. We then describe our original implementation. Finally, we describe its successor, which is the new implementation based on binary decision diagrams.

3.1. Algorithm for choosing new parameter values

The performer module is responsible for choosing new values for the music system variables in real-time, during a musical performance. In the following, we use the term *configuration* to refer to a particular set of values taken on by the music system variables and custom variables; and we refer to the set of configurations allowed by the rules as the *set of allowed configurations*. When an update of the variables is required, the following steps are taken:

I. Values of the uncontrollable variables (those not used to control the music system, but which describe external musical processes) are read in. Then the set of allowed configurations is reduced so that it includes only those configurations in which the uncontrollable variables have these values.
II. The variables which have associated VMMs are iterated over in order of their user-defined priorities. For each one (i) a value is chosen from the VMM, and (ii) the set of allowed configurations is reduced so that it only includes those configurations in which the variable has the newly chosen value. If there are no allowed configurations in which the variable has this value, the set of allowed configurations is not changed (i.e. the VMM-chosen value is ignored).
III. A random configuration is drawn from the remaining set of allowed configurations, and the values of the controllable music system variables are output.

3.2. First implementation

In the following, we describe how these steps were implemented in the performer module used in the first version of the Agent Design Toolkit. This will serve to introduce some key concepts and motivate the new implementation described below.

In the first version of the toolkit, the set of allowed configurations at the beginning of step (I) was represented by the set of rules output by the association rule learning algorithms (see previous section). Together, these rules form a *constraint satisfaction problem* (CSP; see e.g. [1]), which is a problem defined by a set of constraints on the values of a set of variables. It is possible to use many types of constraints to define a CSP including the *implies* rules discovered by ARL algorithms and other types required to define custom variables (see previous section). In step (I), when the values of the uncontrollable variables were read, they were encoded as additional constraints in the CSP.

During step (II) values are chosen for the parameters with which temporal models have been associated. Each time a value is drawn from a temporal model, it is encoded as a constraint and added to the CSP. Before continuing, the CSP is tested to see if it is feasible (i.e. at least one solution exists), since it is possible that a value has been drawn for a parameter, which does not satisfy one or more constraints (thus constraining the parameter to have this value would make the CSP infeasible). If the CSP has become infeasible, the new constraint is removed and no constraint is added for that parameter. If not, the new constraint is allowed to remain. A CSP can be tested for feasibility—and solved, if a solution exists—using a *constraint solver*: a piece of software designed to solve CSPs. The first version of the performer module was implemented in Java and a constraint solver Java library called *Choco* (www.emn.fr/z-info/choco-solver) was used.

Finally, in step (III) a random solution is drawn from the CSP comprising the constraints corresponding to the rules found by the ARL algorithms, as well as those corresponding to the values of the uncontrollable variables, and those corresponding to the values drawn from the temporal models. In the first version of the performer module, this was done by using the *Choco* library to find *all* of the solutions to the CSP and then choosing one randomly.

There were two reasons for our use of a constraint solver for the first implementation of the performer module. First, the rules discovered by ARL algorithms are standard constraints which can be directly used in a constraint solver with low development overhead. Second, general purpose constraint solvers, including *Choco,* support a wide variety of constraint types. The custom variables are implemented using different constraints, and in the early development of the Agent Design Toolkit, it was unknown which custom variables would be included. Thus, it was advantageous to allow for many different possibilities by using a general purpose constraint solver.

We have found that the implementation based on the *Choco* constraint solver generally works well with agents designed to control a small number of variables, or a set of variables with small domains. However, it has a serious flaw which arises from the fact that it is generally impossible to predict in advance (i) how long it will take to solve a CSP and (ii) how many solutions it will have (and how much computer memory will be required to store them). This means that at any time during a musical performance, a CSP may arise for which a random solution cannot be found within the time limits set by the real-time requirement. This is more likely with an agent designed to control many variables, or variables with large domains, since the number of possible variable configurations grows combinatorially. Nevertheless, the key issue is that it is generally impossible to predict when a problematic CSP will arise, and this makes the constraint-based solution unsuitable for live music performance.

In the next subsection, we describe an alternative way to implement steps (I)-(III) above. It involves representing the CSP as a binary decision diagram. This is a data structure which makes it possible to find random solutions to CSPs in a predictable length of time, without requiring all of the solutions to be stored.

3.3. Binary Decision Diagrams

Binary decision diagrams[1] (BDDs) are representations of Boolean functions [11] and in this section, we show how they can be used for real-time musical decision-making. To make use of a BDD, we first transform our CSP into a *Boolean satisfiability* problem (referred to as a *SAT problem*). This can be understood as a CSP in which all of the variables are Boolean; they can only take on values of *true* or *false*. A SAT problem is equivalent to a Boolean function (i.e. a function of Boolean variables) that gives an output of *true* if all the constraints are satisfied and *false* otherwise.

Before continuing, and with reference to our reasons for using a general purpose constraint solver in the first implementation of the performer module (see Section 3.2), we note that not all constraints supported by a general purpose constraint solver can be readily translated to SAT. However, at the time of implementing the BDD-based version of the performer module, the set of custom variables had been defined so it was known that the required constraints could be translated to SAT by available tools.

Once the Boolean function representing the original CSP has been created, it can be transformed into a BDD, which represents the Boolean function as a *directed-acyclic graph* [4]. This two-step process to compute the BDD (CSP to SAT, SAT to BDD), can be performed offline (i.e. before step (I) above). Once it has been completed, it is possible to perform steps (I)-(III) very efficiently and in a predictable length of time. This is because the BDD has special properties which make it a very attractive representation in the context of real-time musical decision-making. Using a BDD, the following operations are possible [5, 11]:

Check if there are any solutions: This is required for step (II) above. Using a BDD, it can be performed in constant time.

Count the number of solutions: This is required for choosing a random solution. It can be done in *O(nB)* time, where *n* is the number of variables in the Boolean function and *B* is the number of nodes in the BDD.

Choose solutions randomly, with all solutions being equally likely: This is required for step (III) and it can be performed in *O(n)* time or less.

Projection: This is the operation whereby the solution space can be restricted to one in which extra constraints are true, such as when the values of uncontrollable variables are read in step (I) and when values are chosen for temporally modelled variables in step (II). It can be performed in *O(B)* time.

All of these operations require the BDD to be constructed to begin with. While the process of transforming a Boolean function into a BDD can be done offline, and therefore is not time-critical, it has two pitfalls which we mention here. First, for a given Boolean function, the size of the BDD (i.e. the number of nodes, *B*) is very sensitive to the way in which the Boolean variables are ordered. A sub-optimal variable ordering can lead to a BDD which is much greater in size than that which would result from the optimal variable ordering. The problem of finding the optimal variable ordering is very hard to solve (it is coNP-complete, in computer science terminology) [11]. However, heuristic algorithms exist which can generally find reasonably good variable orderings. The second pitfall is that some Boolean functions simply cannot be compactly represented using a BDD, even if the optimal variable ordering is known [11]. However, these problems have not prevented BDDs from being successfully used in many different problem domains.

3.4. BDD-based implementation

We have implemented a BDD-based performer module as a plugin for the Max interactive platform (www.cycling74.com). It is written in C++. In this section, we present the implementation details and details of the third party software libraries on which our implementation is built. As described in Section 2.4, the output of the Agent Design process comprises three parts: descriptions of the variables, including the custom variables; VMMs for certain variables; and a set of rules describing the dependencies between variables. The core functionality of performer module is to performs steps (I)-(III) above. To do this it requires a BDD representing the CSP corresponding to the rules.

The construction of the BDD from the CSP is a two-stage process, as described in Section 3.3. The first stage of this process (CSP-SAT) is performed by the Agent Designer, using the Java-based *Sugar* CSP library (http://bach.istc.kobe-u.ac.jp/sugar/) [22]. The SAT

[1] We use this term as it is generally used in the literature to mean *reduced, ordered* binary decision diagrams [11].

representation, along with the agent behaviour specification, are loaded into the performer module. The performer module is based on the *CUDD* package (http://vlsi.colorado.edu/~fabio/CUDD/) for creating and manipulating BDDs (as well as other decision diagrams). This is used to convert the SAT representation into a BDD at load-time, and to perform the BDD projection operations and find random solutions at performance time. Though the conversion from SAT to BDD can in theory take a long time—and so might better be done as a separate, offline process so as not to delay the loading of an agent—we have not found it to take very long in practise (see next Section).

4. PERFORMANCE OF THE BINARY DECISION DIAGRAM

In the previous section, we argued that a performer module based on a constraint solver was unsuitable for live music performance due to its unpredictability. We proposed that an implementation based on BDDs is superior because the resources (time and computer memory) required during performance are more predictable. The uncertainty associated with the BDD-based implementation lies in the conversion from a CSP to a BDD which can result in a BDD which is too large to solve in real-time. However, it will be known at design time (rather than performance time) if an agent cannot be used, and if this is the case, steps can be taken to alter the agent or find another course of action. Furthermore, since BDDs have been used in many other applications to greatly improve the speed at which constraint problems can be solved, we have reason to believe that not only will the BDD-based performer will be more predictable, but it will often be much more efficient as well. In this section we present results which support these arguments.

To compare the performance of the BDD-based performer module with the constraint solver based one, repeated measurements were made of the time taken by each implementation to update the values of sets music system variables. Three different agents were used: they were (i) an agent for performing electronic music, described in [14]; (ii) an agent for performing improvised electroacoustic music, described in [16]; and (iii) an agent for performing drum and bass music, also described in [16]. The number of variables and rules associated with the agents are given in the second and third columns of Table 1. Each performer module performed with each agent for 1000 variable updates, and the times taken to perform the update calculations were recorded. The computer used for all measurements was a 2007 model Apple Macbook Pro with a 2.4 GHz CPU and 4 GB of RAM.

For each set of 1000 measurements, we calculated the normalised standard deviation (i.e. the standard deviation divided by the mean) to show the spread of times taken in a way that is independent of the time values themselves. This gives a good indication of the predictability of the time required to perform a variable update. In addition, we calculated the mean time taken across all 1000 measurements, to compare the efficiency of the two implementations.

The results for each agent as well as the mean across all agents, are shown in Table 1, columns 4-7. Clearly, the time taken for parameter updates is more predictable using the BDD-based performance module for which the mean normalised standard deviation of the time per update was 0.13 compared 1.33 for the constraint solver based implementation. To further illustrate this, we include Figure 2, which shows the normalised update times for 100 of the 1000 updates performed by each of the two performer modules using the electronic music agent. In addition, the results show that for each of the agents used, the BDD-based performance module was much faster than the constraint solver based one (a mean time per update of 0.08 ms compared to 75 ms).

We have not encountered a CSP which resulted in a BDD too large for real-time performance. The BDD sizes (numbers of binary variables and numbers of nodes) are given in Table 1, for each of the three agents used for this experiment. Also given in the time taken to create the BDD (averaged over 10 trials). The most complex agent was the electronic music agent which was used to control 102 music system variables (the number of variables given includes custom variables). It resulted in a BDD with 8262 nodes which took approximately 14.7 seconds to create from the SAT representation. Our results do not show the average time per update growing with the BDD size and number of variables. This can be attributed to the other variations between agents, such as the number of VMM-modelled variables which affects the number of BDD operations that must be performed.

Agent	# Vars	# Rules	Normalised Std. Dev.		Mean time per update (ms)		# Binary Variables	# BDD Nodes	BDD Creation Time (s)
			Con	BDD	Con	BDD			
Electronic [14]	103	500	1.5	0.1	175	0.06	226	8262	14.76
Electroacoustic [16]	13	26	1.3	0.2	25	0.08	36	1040	0.002
Drum and Bass [16]	14	18	1.2	0.1	25	0.1	32	291	0.001
		Mean	**1.33**	**0.13**	**75**	**0.08**			

Table 1. Comparison between the BDD-based performer module (BDD) and the constraint solver based performer module (Con) for three autonomous agents. See text for details.

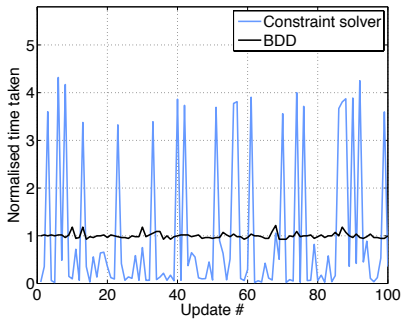

Figure 2. Normalised time taken to update variable values for the constraint solver and BDD-based implementations.

Finally, we note an additional problem encountered using the constraint solver based performer module when running the electronic music agent. On occasion, the number of solutions to the CSP was too large and they could not be stored in memory. This meant that the software could not perform the update. For the timing measurements shown above, restrictions were added to the agent specification to prevent this occurring. That the BDD-based implementation allows random solutions to be found without calculating and storing all solutions is an important advantage.

These results are consistent with BDD-theory, which says the update time required by the performer module only depends on the BDD size and the number of variables, which do not change during a musical performance. This contrasts with the constraint solver based implementation for which the update time depends on the number of CSP solutions, which can change from one update to the next. The results also show that for the agents used—which are representative of the agents produced by the Agent Designer Toolkit to date—our BDD-based performer module is much more efficient than the constraint solver based one. Some of this increased performance can be attributed to the inherent speed difference between Java and C++. In addition, there are constraint solvers which are faster than *Choco*. However neither of these factors accounts for the size of the performance improvement, particularly for the electronic music agent.

5. DISCUSSION

Considerable research has been done into the application of constraint programming (i.e. the use of CSPs and constraint solvers) to modelling music theories, both traditional and novel (see [2] for a survey). Much of this work has focussed on ways to represent music (i.e. a musical score) so that constraints of sufficient sophistication and musicality can be used. In comparison, the block-based representation of musical data used in the Agent Design Toolkit is very simple. However, in our case constraints are learnt whereas in the work surveyed in [2], they are specified by the user. Thus, in the Agent Design Toolkit, any constraint-related capability must be paired with a corresponding machine learning capability and the machine learning problem of discovering patterns as complex as those found in the theory of classical harmony, for example, is an extremely difficult one. Our use of custom variables (see Section 2.3, [14] and [16]) is one way in which a musician can increase the sophistication and musicality of the representation and thereby enhance the ability of the machine learning algorithms to find musically salient patterns and relationships.

There has been some research into the use of Constraints in real-time, interactive applications [3, 17], but only with manually specified constraints on musical data. As previously mentioned, it is impossible to know in advance, how long a constraint solver will take to find a solution. Thus, in [3] where a constraint solver is used in a real-time scenario, the workaround is to use a timeout which stops the search if a solution cannot be found within a specified duration. In contrast, the predictability of our BDD-based solver is a great advantage in real-time applications.

5.1. Future work

To improve the performer module, we plan to investigate alternative ways to deal with the possibility that a VMM-chosen value might not be consistent with the association rules. For example, a method has been proposed for adjusting the probabilities in a Markov model to account for external constraints while still remaining consistent with the model [18].

In the context of the Agent Designer Toolkit as a whole, we plan to carry out a thorough evaluation in an upcoming series of user studies. In addition to usability

testing, we will evaluate the toolkit in its capacity as a *creativity support tool* [21] and the extent to which it is useful to computer music practitioners in their creative work relating to agent design and automation. To this end, we have already integrated the toolkit into the Ableton Live music production software [16].

6. CONCLUSION

Constraint based systems are a powerful way for musicians to make musical decision-making agents. Constraints provide a way of formally specifying a set of relations between musical elements that can accurately capture a musician's conceptualization of their music. However, CSPs are not generative models, meaning that although they provide an accurate representation of a set of musical constraints, they cannot be used to derive musical decisions. In addition, they can be difficult to solve in a real-time context. The BDD provides a real-time generative representation of a CSP an a way that is practically useable by musicians. To our knowledge, this is the first report on the use of binary decision diagrams in a real-time music application.

7. ACKNOWLEDGEMENTS

We would like to thank Nina Narodytska and Toby Walshe for their thoughtful and helpful advice.

8. REFERENCES

[1] Apt, K.R. *Principles of Constraint Programming*. 2003, Cambridge University Press.

[2] Anders, T. and Miranda, M.R., "Constraint programming systems for modeling music theories and composition", ACM Comput. Surv., 2011. 43(4): pp. 30:1-30:38.

[3] Anders, T. and Miranda, M.R., "Constraint-based composition in realtime", in Proc. ICMC, Belfast, Northern Ireland, 2008.

[4] Bryant, R.E., "Graph-based algorithms for Boolean function manipulation", IEEE T. Comput., 1986. C-35(8): pp. 677-691.

[5] Darwiche, A., Marquis, P., "A knowledge compilation map", J. Artif. Intell. Res., 2002. 17: pp. 229-264.

[6] Drummond, J., "Understanding interactive systems", Organised Sound, 2009. 14(2): pp. 124-133.

[7] Fails, J.A. and Olsen, Jr., D.R., "Interactive Machine Learning", in Proc. International Conference on Intelligent User Interfaces, Miami, USA, 2003, pp. 39-45.

[8] Fiebrink, R., Cook, P.R. and Trueman, D., "Human model evaluation in interactive supervised learning", in Proc. CHI, Vancouver, Canada, 2011, pp. 147-156.

[9] Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning, 2nd Edition*. 2009, New York: Springer.

[10] Hsu, W., "Two approaches for interaction management in timbre-aware improvisation systems", in Proc. ICMC, Belfast, Northern Ireland, 2008.

[11] Knuth, D.E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. 2011, Upper Saddle River, New Jersey: Addison-Wesley.

[12] Lewis, G., "Too many notes: Computers, complexity and culture in voyager", Leonardo Music J., 2000. 10: pp. 33-39.

[13] Martin, A., Jin, C., van Schaik, A. and Martens, W.L., "Partially observable Markov decision processes for interactive music systems", in Proc. ICMC, New York, USA, 2010, pp. 480-493.

[14] Martin, A., Jin, C.T. and Bown, O., "A toolkit for designing interactive musical agents", in Proc. OZCHI, Canberra, Australia, 2011, pp. 194-197.

[15] Martin, A., McEwan, A., Jin, C.T. and Martens, W.L., "A similarity algorithm for interactive style imitation", in Proc. ICMC, Huddersfield, UK, 2011, pp. 571-574.

[16] Martin, A., Jin, C.T., Carey, B. and Bown, O.. "Creative Experiments Using a System for Learning High-Level Performance Structure in Ableton Live", in Proc. SMC, Copenhagen, Denmark, 2012.

[17] Pachet, F. and Delerue, O., "Midispace: a temporal constraint-based music spatializer", in Proc. ACM International Conference on Multimedia, Bristol, UK, 1998, pp. 351-359.

[18] Pachet, F., Roy, P. and Barbieri, G., "Finite-length Markov processes with constraints", in Proc. IJCAI, Barcelona, Spain, 2011, pp. 635-642.

[19] Ron, D., Singer, S. and Tishby, N., "The power of amnesia: Learning probabilistic automata with variable memory length", Mach. Learn., 1996. 25: pp. 113-149.

[20] Rowe, R. *Interactive music systems*. 1993, Cambridge, MA: MIT Press.

[21] Shneiderman, B., "Creativity support tools: accelerating discovery and innovation", Commun. ACM, 2007. 50(12): pp. 20-32.

[22] Tamura, N., Taga, A., Kitagawa, S. and Banbara, M., "Compiling finite linear CSP into SAT", Constraints, 2009. 14: pp. 254-272.