The Resource-Instance Model of Music Representation[1]

Roger B. Dannenberg, Dean Rubine, Tom Neuendorffer
Information Technology Center
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
Email: dannenberg@cs.cmu.edu

ABSTRACT

Traditional software synthesis systems, such as Music V, utilize an *instance* model of computation in which each note instantiates a new copy of an instrument. An alternative is the *resource* model, exemplified by MIDI ‘‘mono mode’’, in which multiple updates can modify a sound continuously, and where multiple notes share a single instrument. We have developed a unified, general model for describing combinations of instances and resources. Our model is a hierarchy in which resource-instances at one level generate output which is combined to form updates to the next level. The model can express complex system configurations in a natural way.

1. Introduction

Two opposing formalisms are prevalent in music representations. In the *resource* model, sounds or notes are produced by controlling an instrument (the resource). In the *instance* model, sounds or notes are considered to be independent and isolated. Resource and instance models can be seen in traditional music notation, computer music scores, score languages, MIDI, synthesis hardware, and synthesis software. Although the distinction between resource and instance models is fundamental, it is not often made (perhaps because the implications of the distinction are not well understood).

Once the distinction is made, it can be seen that virtually every music representation system exhibits *both* formalisms. In other words, music representations have aspects of both the resource *and* instance models. Furthermore, these seemingly mutually exclusive models can be combined to create a comprehensive formalism. Armed with this new formalism, we can shed new light on existing representation schemes, exposing hidden assumptions, revealing subtle ambiguities, and unmasking limitations.

We will begin by explaining the instance and resource models in greater detail. We then describe our new formalism, which integrates the two models. The new ‘‘resource-instance’’ formalism is then applied to MIDI and Music V to illustrate particular characteristics of these representation systems. Then, we describe how we are applying the formalism in a new system for music representation and synthesis.

[1]Published as: Dannenberg, Rubine, and Neuendorffer, ‘‘The Resource-Instance Model of Music Representation,’’ in Proceedings of the 1991 International Computer Music Conference, International Computer Music Association, (October 1991), pp. 428-432.

2. The Resource Model

In the resource model of music representation, there are sound sources (*resources*) that are controlled by *updates*. Updates include continuous functions such as a vibrato contour and discrete events such as ‘‘note on’’. Updates are combined and presented to the resource, which can produce only one sound at a time.

A familiar example of the resource model is MIDI Mono Mode [IMA 89], in which all MIDI messages are presented to a single tone generating resource. Discrete updates such as ‘‘note on’’ messages change the pitch (they do *not* invoke a new sound using another tone generator), and continuous controls such as modulation or pitch bend can also update the tone generator. Mono Mode applies to a channel, so the channel number becomes the name for the resource, and updates are addressed to the resource via their channel designation.

The resource model is most often encountered when there are physical resources that must be considered such as synthesizer hardware modules or acoustic instruments. In software synthesis, it is possible to fabricate ‘‘virtual’’ instruments almost without limitation, so it is possible to ignore resources completely. Even in traditional music making with acoustic instruments, composers can often ignore the question of whether the first or second violin produces a note. This line of thinking leads to the instance model.

3. The Instance Model

In the instance model of music representation, the resource that produces the sound is not considered, and only the attributes of the sound are relevant. It is as if, for each sound, we create a copy (or *instance*) of a resource to produce each sound. In the instance model, all sounds are independent, and there is no limitation on the number of simultaneous sounds that can occur.

Music V [Mathews 69] is an example of the instance model. For each note in the score language, a software instrument is instantiated to generate the sound. After the sound has been generated, the software instrument is deleted. In the instance model, it is well-defined to generate two or more sounds that are exactly simultaneous and identical in all respects. In contrast, in the resource model, two ‘‘identical’’ notes would at least require generation by distinct resources, so these notes could not be truly identical.

4. The Resource-Instance Model

The resource and instance models cannot be pushed very far before they break down. In most resource-oriented systems, there are multiple tone generators, be they violinists or register sets in multiplexed hardware. This gives rise to the concept of ‘‘limited polyphony,’’ which amounts to saying the instance model is supported up to a certain level of polyphony by managing a pool of undifferentiated resources. We see this even within acoustic instruments: a guitar has 6 strings, allowing a limited implementation of the instance model. An important aspect of transcriptions for classical guitar is the careful use of a limited number of strings and fingers to implement music that is instance-model oriented.

Instance models also break down rapidly upon close inspection. In particular, the independent isolated sounds described by the model must be combined. In the ideal world of mathematics, we can say that (due to linearity) superposition holds, and combination is not interesting. In practice, the fact that a mandolin has double strings tuned in unison indicates that the situation is not so simple. Also, we know that the generation of a sound is often just the beginning of the music-making process. Sounds can be assigned to channels of a mixer, tracks of a tape, or passed through (non-linear) effects processors. To describe the big picture, we are forced into thinking about resources.

The *resource-instance* model uses both resources and instances to address these problems. Consider the case of two electric guitars with distortion boxes playing through a single sound system. The resource-instance model describes this case as follows: the sound system is a resource. Resources receive and process updates; in this case the updates are the sounds produced by two distortion guitars (we are deliberately stretching the intuitive concept of ‘‘update’’ here). The processing of the updates is to form their sum. The guitars are instances of a complex instrument. Each instance consists of a distortion box (a resource) fed by 6 strings (updates to the distortion box). The string updates are processed by adding them and then applying distortion.

The term *update* deserves special attention. An update in this model is any input that controls a resource. As described in section 2, an update can be continuous or discrete, but most importantly, an update can be generated by another resource. Thus, updates and resources form a hierarchy.

Note that from the point of view of the strings, the distortion box is a resource, but from the point of view of the sound system, distortion boxes are instances. Similarly, strings are instances from the point of view of the distortion box, but strings are also resources. Each of the 6 strings receives updates such as fretting, releasing, and strumming. These updates are not combined by simple addition but by some complex response.

It seems clear now that the concepts of resources and instances are necessary in a complete model, but that whether something is a resource or an instance depends on the point of view. Furthermore, there are hierarchical relationships formed when the sounds produced by instances are combined at a resource. The *resource-instance* model represents sound production as a hierarchy. The hierarchy can be represented statically, as in figure 1, which shows schematically how resources are connected. Each resource in the static figure corresponds to one or more instances as shown in the dynamic representation shown in figure 2.

The resource-instance model does not dictate when resources are created. The guitar example implied that the guitars and strings were all fixed in number. However, the instance model can be accommodated by instantiating a new guitar for each note. A special form of update is the *create-instance* update, whose destination is the resource below the instance in the hierarchy. To play a guitar note under the instance model, one would send a create-instance update to the sound system to obtain a guitar, send a create-instance update to the guitar to obtain a string, and send updates to the string to play the note. The resource created by a create-instance update is an instance of a *prototype*. In our terminology, the Music V orchestra language defines prototypes for resources, and a new resource is created for each note in the score. Every resource is an instance of some prototype.

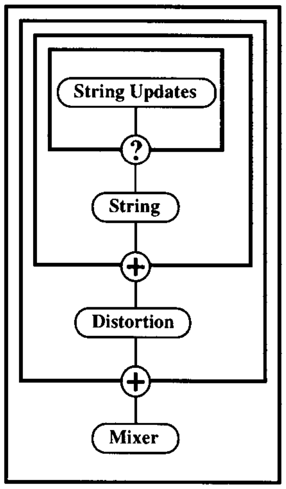

Figure 1: Static description of (a schema for) a resource-instance hierarchy, showing prototypes of resources and how updates are combined. This is essentially a ‘‘patch’’ diagram.

5. Explicating Existing Representations

The resource-instance model can help to illustrate details of various representations. For example, in the model, every MIDI message is an update of some kind. We can then ask, what resource is the update for? Is a note-on message an update to the channel to create a new note, or does every channel have 128 pitch resources which are the destinations of note-on’s? It turns out that MIDI is ambiguous on this point, and manufacturers have not agreed on the answer. On some synthesizers, a second note-on to the same channel and pitch causes the note to retrigger. Clearly, these synthesizers consider the note-on to be an update to a pitch resource. Other synthesizers will actually produce two notes on the same channel with the same pitch. In these ‘‘instance model’’ synthesizers, the target of the note-on is the channel, not a pitch resource. This creates yet another ambiguity: since MIDI has no names for note instances other than pitch, there is no way to designate the destination for a note-off update. (This is not a problem for the pitch-as-resource synthesizers, where the destination is always the pitch resource.)

In Music V, it is not actually the case that notes must be independent. By sharing buffers between instruments, it is possible to construct a hierarchical structure in which updates may be directed to multiple levels. This is how one would combine notes before sending them to a reverberation unit, for example. However, there is no way to properly implement the create-instance update. One cannot support an arbitrary number of instances of reverberation units, for example.

These examples illustrate how the resource-instance model can shed light on subtle aspects of the design of music representations.

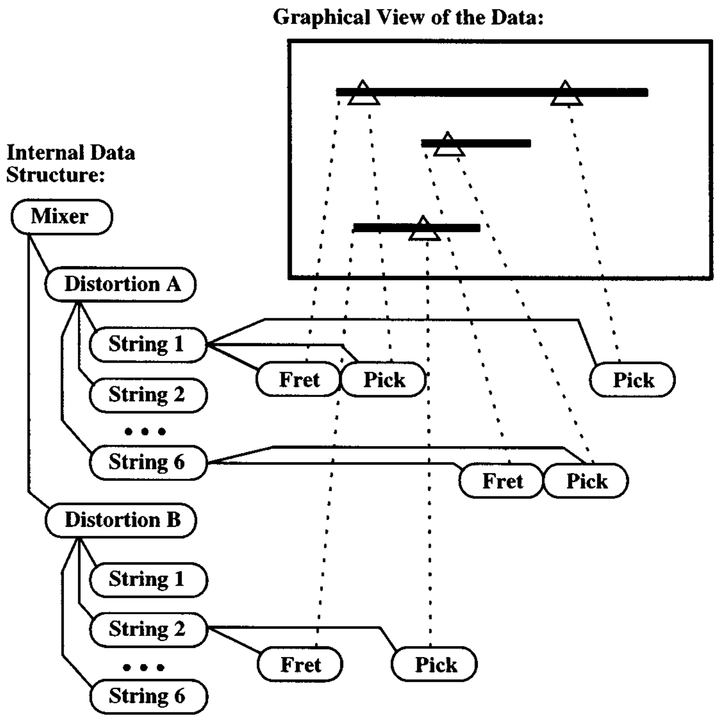

Figure 2: A dynamic representation of figure 1 in which resources are instantiated to form a tree. Also shown is a graphical view for use in editing the structure.

6. Relation to Object Oriented Programming

This model bears some resemblance to object oriented programming (OOP) models, but there are important differences. Our resources are like objects in that each resource is unique and can be referenced. Resources are instances of a prototype, just as objects are instances of a class. In our model, we can simulate instance models by creating a new resource for every update. In section 4 we explained how a new guitar could be instantiated for every note to obtain instance-model guitars. The OOP language Smalltalk [Goldberg 83] uses a similar technique with numbers: a number is an object, but every operation, such as addition, creates a new instance of the number class to represent the resulting value.

In spite of these similarities, our model contains specific ideas that are not part of the OOP model. OOP does not include the notions of outputs and updates. Although one can build object hierarchies with OOP, this is not part of the OOP model. Furthermore, OOP does not have combining operations or a static description of a resource-instance hierarchy as in figure 1. (The class hierarchy of OOP is unrelated.)

7. Supporting the Resource-Instance Model

We are developing an integrated music workstation that supports the resource-instance model. In fact, we developed the model during the design of a patch editor. A number of patch editors have been described in the literature [Desain 86, Helmuth 90], but it bothered us that these editors do not address the question of resources. This led to the resource-instance model and representations such as figure 1, which is what an instrument designer might create with our editor. Our patch editor will provide two novel capabilities: first, the levels of hierarchy in the resource-instance model must be represented. (We plan to use enclosing boxes to group the unit generators that constitute a resource prototype from which instances are made.) Second, the operators for combining the outputs from instances must be explicit. Summation of all outputs will be the common case, but other possibilities exist. For example, the *replacement* combination takes only the most recent update and ignores all others.

Scores also have a hierarchical representation in this system. Notes do not exist in isolation as in the instance model, but rather are updates to resources. A note becomes a resource to which updates (pitch, modulation, etc.) may be directed. The score view in figure 2 shows how string updates might be represented graphically. We are implementing a visual score editor based on an earlier multiple-hierarchy data structure for music representation [Dannenberg 90]. Designing a user interface in which this hierarchy seems natural and automatic is a challenge we still face.

We are extending our editor to handle multiple media. Here, we see similar issues of instances and resources. For example, how does one represent updates in an animation? An animation might have multiple instances of people, each with instances of limbs. One might even want to synchronize multiple animations. The resource-instance model provides a natural solution to the naming and representation issues raised here.

8. Summary and Conclusions

The *instance* model of computation, in which each note instantiates a new copy of an instrument, and the *resource* model, in which multiple updates can modify a sound continuously, often exist together at multiple levels of a hierarchy. We have developed a unified, general model for describing combinations of instances and resources. The *resource-instance* model can express and help to understand complex system configurations in a natural way. The model has practical applications in the following areas: (1) the description of synthesis algorithms, (2) the design of patch editors, (3) the extension of orchestra languages to incorporate the resource model, (4) the design of score representations and editors, and (5) the design of synthesis hardware and software.

More importantly, we feel the model provides a frame of reference that can help to understand and compare various music representation systems. The model is being used in and supported by the design of a digital audio workstation.

9. Acknowledgments

This work was supported by a grant from the IBM Corporation.

References

[Dannenberg 90] Dannenberg, Roger B. A Structure for Efficient Update, Incremental Redisplay and Undo in Display-Oriented Editors. *Software: Practice and Experience* 20(2):109-132, February, 1990.

[Desain 86] Desain, P. Graphical Programming in Computer Music, a Proposal. In P. Berg (editor), *Proceedings of the International Computer Music Conference 1986*, pages 161-166. International Computer Music Association, 1986.

[Goldberg 83] Goldberg, A. and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

[Helmuth 90] Helmuth, M. PATCHMIX: A C++ X Graphical Interface to Cmix. In *Proceedings of the 1990 International Computer Music Conference*, pages 273-275. Computer Music Association, 1990.

[IMA 89] IMA. *MIDI 1.0 Detailed Specification.* International MIDI Association, Los Angeles, CA, 1989.

[Mathews 69] Mathews, M. V. *The Technology of Computer Music.* MIT Press, Boston, 1969.

Table of Contents

List of Figures