This section takes us from the works that use the c5 agent-toolkit to a new toolkit, called Diagram. *It introduces the first artwork to exploit this framework —* Loops Score, *the music for* Loops. *Diagram is a loose and interacting set of extensions to c5 that are designed to ameliorate the apparently persistent problems that artists employing complex agents face. In the technical discussion that follows, I respond to my previous critique of c5 and its use in* alphaWolf. Diagram *is also the set of technologies that will lead to* how long....

Chapter 6 — The Diagram framework & *Loops Score*

In the previous description of the complex, multi-programmer project *alphaWolf*, we discovered a number of inefficiencies in the way that it was assembled. While some of these problems may have stemmed from what one might call "impedance mismatches" between the components used to assemble *alphaWolf*, it seems more likely that the failings and weaknesses of this large assembly ideas from the c5 agent toolkit used for that work were more infrastructural than tied to any particular algorithm or representation. Both *Loops* and *The Music Creatures* dodged or postponed many of the issues identified — simply because of the size or goals of the agents involved, or the technologies deployed around their creation.

For the two works for dance theater that close this thesis — *how long...* and *22* — we had the great fortune of having two and a half years of notice before premiering the works. I could have spent all of this time constructing new fragments — new action-selection techniques, new classifiers for the perception system and new representations for pose-graph motor system. Instead I took some time to consider how these fragments were being assembled and the"glue" systems that hold the other systems together.

1. Complex assemblages – the inversion (of the inversion) of control

The scope of these"glue systems" is both a little broader that the influential"design patterns" of software engineering and substantially narrower than a complete, integrated, academic AI system. Broader —for they are more concrete and more multiply instantiated than the abstract design pattern; Narrower — for they make no claim to be a complete or even partial solution to any particular AI *world* or *problem domain* by themselves.

The so called"gang-of-four" design patterns book — **E. Gamma, R. Helm, R. Johnson, J. Vlissides**, *Design Patterns, Elements of reusable object-oriented software*. Addison-Wesley, 1995.

Primary texts on software engineering's ideas concerning the "Inversion of Control" are hard to come by. An overview of a broad variety of"framework integration problems" can be found in: **M. Mattsson, J. Bosch, M. E. Fayad**, *Framework integration problems, causes, solutions*. Communications of the ACM, 42 (10). 1999.

Online resources abound, however:

The Apache project's Excalibur project: http://excalibur.apache.org/

the "HiveMind" project at Apache Jakarta: http://jakarta.apache.org/hivemind/ioc.html

The PicoContainer framework: http://www.picocontainer.org/

The Spring framework: http://www.springframework.org/

The inability to draw a stable, hierarchical diagram of either control, instantiation, execution ordering or signal flow has been seen before in both published descriptions of hypothetical AI systems and throughout software engineering. We have seen it, in miniature, in our analysis of the source files of *alphaWolf, page 84*; we have seen it in the odd inversion of motor-system outside the colony of *Loops, page 101*; and we have seen it in the ad hoc reinterpretations of the perception / action / motor decompositions of *The Music Creatures*.

Indeed, historically, the very idea of the software agent has been motivated by the problem of creating heterogeneous assemblages of interdependent modules (agents) to solve complex tasks in complex domains— be they economies or soccer games. Here the autonomy of the agent aligns again with the tractability of the decomposition of the complex task into interacting parts.

The tendency here, in both micro- and macro- conceptions of the problem, is to simplify the interconnectivity between these modules (agents). Hallmarks of this trend are extremely diverse, but one might re-read blackboard systems, artificial biochemistries, and various instantiation languages for behavior systems as searches for a minimal but powerful "glue" for signal flow, execution control or system instantiation. Each, I believe, demonstrates in insistence on the expressive role of the ubiquitous system diagram in AI's "system paper", with its complex boxes and thin arrows, a desire for "modularity," re-articulated repeatedly.

Blackboard architectures have a long history both inside and outside the agent. For a software engineering perspective: **F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad,** and **M. Stal.** *Pattern-Oriented Software Architecture: A System Of Patterns.* West Sussex, England: John Wiley & Sons Ltd., 1996. Inside the agent: **B. Hayes-Roth,** *A Blackboard Architecture for Control,* Artificial Intelligence, 1985.

Synthetic biochemical communication: **S. Grand, D. Cliff, A. Malhotra,** *Creatures: artificial life autonomous software agents for home entertainment.* Proceedings of the first international conference on Autonomous agents, ACM, 1997.

Inside purer software engineering pursuits, similar problems and solutions are devised and re-devised. Most prevalent are the problems surrounding the instantiation of complex assemblages, and there is a thread of solutions that are typically referred to as the "inversion of control" or more tersely, IoC.

XML is a ubiquitous W3C committee standard — http://www.w3.org/XML/

Many IoC systems propose a separate instantiation language (typically XML) for all material that is written in some other language — one programming technique for the boxes, another for the arrows. Still more IoC systems provide registry and retrieval mechanisms for objects to use in order to find the other objects that they ought to connect to — a central place for boxes to find their arrows.

The goal in both cases is to *decouple* modules from each other, indeed, to keep modules modular, the boxes boxed-up and the arrows lightweight. The dreams of the modular cure many of the things that were so hard to maintain during the development of *alphaWolf* — separation of effect, incremental "testability", a late binding reconfigurability and the ability to reuse pieces of a work in the next.

IoC systems that use a separate configuration language, commit themselves to two positions. Firstly that the glue between systems is necessarily simpler than the systems being glued together — that the box is bigger than the arrow —, and that this gluing does not require or deserve the complexities of a "full strength" programming language nor the environments that accompany them — that the organization of arrows is simpler than the contents of the box. Secondly that this assemblage is separated both in time and (metaphorically) in space from the cores of the modules themselves — these systems talk of a "design time", where the arrangement of modules is decided upon prior to execution, and a design space outside the modules.

Indeed, IoC solutions in general fail to allow the *inversion* of their particular inversion of control — or, less rhetorically, they fail to be particularly sophisticated about where the control (instantiation, execution or communication) ends up after it is taken away from the insides of the module. These systems are not failing their problem domain, but the challenges that they have been designed to face simply are not as complex as the architectural problems fundamental in making complex agents. The idea, therefore, that a module might, during execution, long after instantiation, reorganize existing modules, construct some anew and partially delete some more, is at best a rather long way from traditional IoC systems' point of departure. To deposit control into a central registry, a configuration file or a set of instantiation descriptions, is a fundamental dilution of the power of a module in the system over the modular; and if our idea of the module is our receptacle for our introspection, our reusability and our extensibility strategies, then this maneuver reinforces the problems of constructing complex assemblages of modules with complex life-cycles.

This of course isn't of much relevance for problems where the connections between modules are a simple affair and the modules themselves provide all of the power, but our agents are already dynamic and getting more so as we move from *Loops* to *how long*.... Perception systems grow, action systems grow, long-term authorship techniques cut across both, motor systems model what actions systems do, action systems set one structure up only to develop another later. Thus, artificial agents do not appear to be in the group of problems that allow the longer-term use of parallel instantiation languages.

It is worth noting in passing that early in the work of the Synthetic Characters group just such an instantiation language was created and used (for the large-scale installations *Swamped!*, 1997 and (void *) — *a cast of characters*, 1998). This design, which was part of the *Scoot* framework, was ultimately superseded and replaced by the current 'c' series of agent toolkits for a number of reasons, not least of which are the arguments presented above. Finally, we note of course that these IoC strategies run parallel to, but are in general more sophisticated than, the virtual wire of environments for digital art — which embody the fantasy of the system diagram — but in all fields the tactics are the same.

B. Blumberg, *Swamped! Using plush toys to direct autonomous animated characters*. Proceedings of SIGGRAPH 98: conference abstracts and applications., ACM Press, 1998.

B. Blumberg, *(void*): A Cast of Characters*. Proceedings of SIGGRAPH 99: conference abstracts and applications. ACM Press, 1999.

It is common to refer to the instantiation phase of an IoC container system as the time when "dependency injection" occurs. This is the time when connections between systems are forged, either pushed to modules from a central description of what the connections should be, or pulled by the modules from a central naming service. In the work that follows we reject the singular nature of the "design time" and look at structures that remain malleable across the life-cycle of the agent, and some structures across the life-cycle of the creation of the artwork.

This project shares, indeed inherits, the broad tactical goal of IoC systems — to find a low number of powerful, tractable, indeed author-able, strategies for controlling, assembling, and ordering the execution and communication of modules. But our approach here differ from previous IoC attempts in that it admits immediately that that "low number" may be in fact greater than one; that the control, assemblage and ordering of modules are intersecting but not identical problems; and that these issues necessarily breach attempts to contain them within the confines of instantiation languages, of "design time" or any other place which is essentially outside modules organized.

For an overview of Aspect-Oriented Programming issues, there is a special issue of the Communications of the ACM: **T. Elrad, R. E. Filman, A. Bader,** *Aspect Oriented Programming : Introduction,* Communications of the ACM, October 2001.

In spirit we are most influenced by the AspectJ project: http://eclipse.org/aspectj

Related to modern IoC systems is the current excitement surrounding Aspect-Oriented programming. AoP, seeks to augment the module vocabulary of object oriented programming to include "concerns" that cut across several classes, instances or methods. Like "mainstream" AoP we are interested in finding alternative connective relationships between modules and alternative authorship strategies for maintaining these relationships. And we will use some of the same techniques that AoP under Java uses — instantiation time, byte-code injection and, most recently, load-time annotations. Unlike AoP we are not looking for generic solutions, but rather very specific ones for the problems that arise while authoring agents.

2. The Context-Tree, a new "working memory" for agents

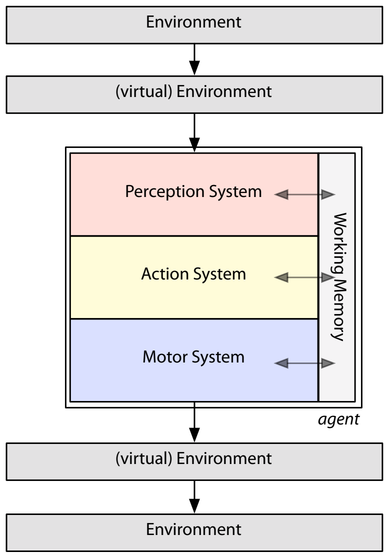


figure 57. The agent decomposed again. According to this diagram "working memory" acts as a conduit between all internal systems.

In the initial description of the c5 toolkit we drew a diagram, a decomposition into perception, action and motor systems connected by a ubiquitous "working memory".

This "working memory" serves as a communication channel, a persistent blackboard where systems could write and read, post and receive messages to each other, while remaining relatively uncoupled. Earlier, I refused to draw real arrows between these boxes — denying the implication that one particular thing flowed in a direction, coupled in a particular way or assumed control over any other. We can continue to refuse to draw arrows, but analyzing how separate concerns inside the agent-toolkit *couple* in these ways is, however, going to be an unavoidable aspect of taming complexities of large agents. In this section we develop a replacement for the simple blackboard-like working memory of c5, called the *context tree.*

The context tree is a tree of execution contexts, or scopes, that loosely follows the execution of the complete code-base involved in a particular work. In its simplest deployment the context tree has the following properties:

- for any given moment during the execution cycle, there is a single special level of the tree (branch or root) that is the "current context".
- each context has a name, and, optionally, any number of children.
- names are not unique throughout the tree, but are unique within a single group of children.
- each context has a parent, bar the top of the tree, the "root context".
- the tree is typically, but not always, fully explored in each execution cycle.
- the mapping from source-code line number to execution context is potentially one-to-many: if a line of code is revisited during execution, it is not necessarily revisited in the same context. Context scope is thus a dynamic scope, rather than something that can be either statically or lexically determined.
- a context has any number of named elements, a mapping from name to object reference, and these are separate from namespace of children contexts. Named elements can be looked up with respect to the current context, should no element be found, successive parent contexts are searched until this element, or *key* is found.
- navigation through the tree, and the creation and deletion of contexts, is explicit.
- the other parts of the context tree can be referenced and searched both relatively and absolutely. The context-tree has aspects of a runtime database.

For information about dynamic scope in Perl — **L. Wall, T. Christiansen, J. Orwant,** *Programming Perl*, O'Reilly, 2000.

For an entry to the debate over Lisp's (early) dynamic scoping: http://en.wikipedia.org/wiki/Scope_(programming)

Clearly my context tree shares many similarities with *dynamic scoping* found, for example, in some dialects of Lisp or more recently Perl. However it differs from these implementations mainly through its explicitness — dynamic scope is not implicitly woven from the language in which the code is written, but rather explicitly navigated, named and used by code that, in this case, may be written in any number of languages. However, given the above description of the context tree, it should be clear that there really isn't very much complexity to it at all. An acceptable simple implementation of the context tree is simply a tree of named hash-maps. As we develop our use of the context tree, we will begin to weaken some of these assumptions: we shall see contexts with multiple parents, and language-level features for accessing and navigating the tree.

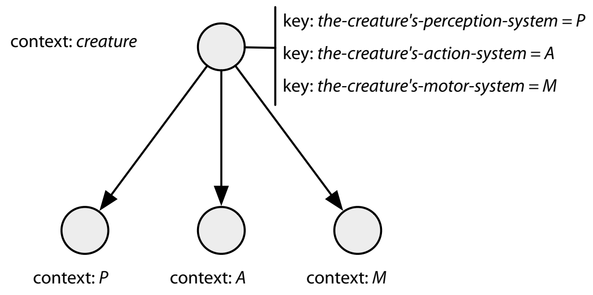

context: *creature*
key: *the-creature's-perception-system = P*
key: *the-creature's-action-system = A*
key: *the-creature's-motor-system = M*
context: *P*
context: *A*
context: *M*

figure 58. The creature context and its three child contexts P, A and M.

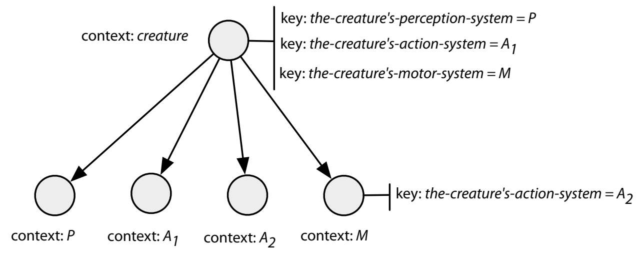
It's reasonably straightforward to see how to turn this tree into a solution for simple inversion of control problems. A creature needs a perception system, an action system and a motor system, and we have a particular instantiated perception system P, a particular action system A, and a particular motor system M; we make a creature-level context creature that contains the key-value bindings *the-perception-system*:P, *the-action-system*:A , and *the-motor-system*:M. We'd like to write this with as little fanfare as possible, in Java (with the "keys" statically imported from definition interfaces):

```
thePerceptionSystem.set(P);
myP = thePreceptionSystem.get();
```

and in Python (using a class specifically designed for context tree access):

```
c.thePerceptionSystem = P;
```

We'll see an even more minimal interface to the context tree below, *page 226*. Typically, Keys like *thePerceptionSystem* are static, that is globally importable and accessible — they obtain that locality and module specificity not through the class-hierarchy-like instance fields but through the context hierarchy. However, later we might see storage that is both instance and context local, *page 204*. In other cases we might pick some other hierarchy structure to present as a tree — in the graphical user interface discussed later, we consider the hierarchy of views as a "context-tree-like" tree, *page 377*. No matter, for having constructed this abstraction, we are free to choose the granularity and the tree to apply it to. For the purposes of this discussion we shall assume the most typical case, that there is a single, statically accessible context-tree shared by the entire runtime system.



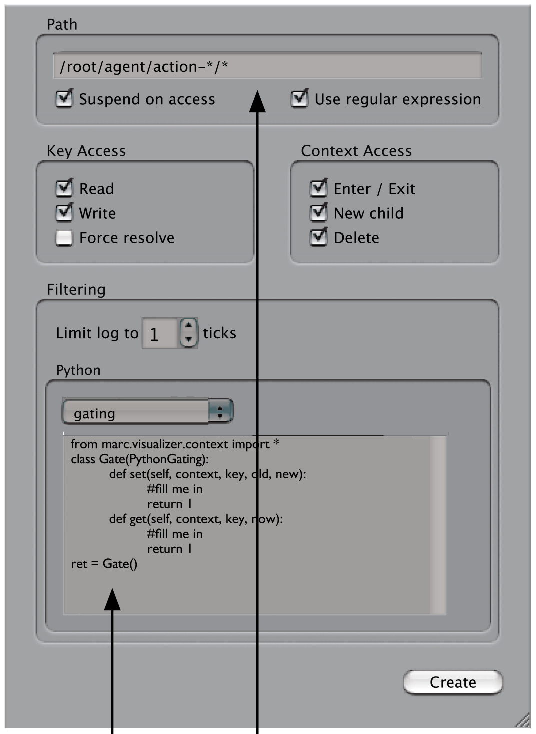
context: *creature*
key: *the-creature's-perception-system = P*
key: *the-creature's-action-system = A_1*
key: *the-creature's-motor-system = M*
key: *the-creature's-action-system = A_2*
context: *P*
context: *A_1*
context: *A_2*
context: *M*

figure 59. One way of ensuring that M refers to A_2 is to write the reference directly into M, locally overriding it for this part of the context tree.

Modules P, A, and M execute in their own, child contexts of creature. When the motor system M needs to find a reference to the action system it looks up *the-action-system* in its own context; although there is no binding there, there is a binding in the parent context creature and action system A is returned. The box thus finds its arrow.

Simple as this example is, there are a few key results to note:

this is a weakly coupling assemblage — this action system is never stored as an module-level variable (for example, it is never stored as an instance variable for any particular object); it can always be obtained from the context tree. The action system can change completely (to the point that the action system is actually a different object reference) without explicitly accessing or notifying any other system. For this to be useful to consumers of this information, context-tree lookup should be fast enough; this is easily achieved by a variety of caching mechanisms that turn a context tree search into a single hash-map lookup in most cases. For this to be useful to authors of systems there should be notification mechanisms that can provide hand-off between changing systems.



Path
Key Access
Context Access
Filtering
Python
Create
run visual elements upon breakpoint access
set breakpoint on XPath and regular context-tree expressions

Path: /root/agent/action-*/*

[✓] Suspend on access [✓] Use regular expression

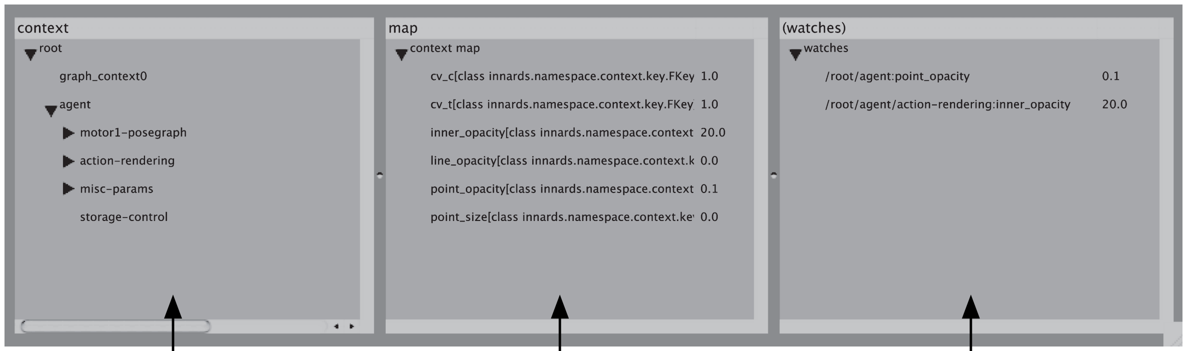
Key Access: [✓] Read [✓] Write [ ] Force resolve

Context Access: [✓] Enter / Exit [✓] New child [✓] Delete

Limit log to 1 ticks

Python: gating

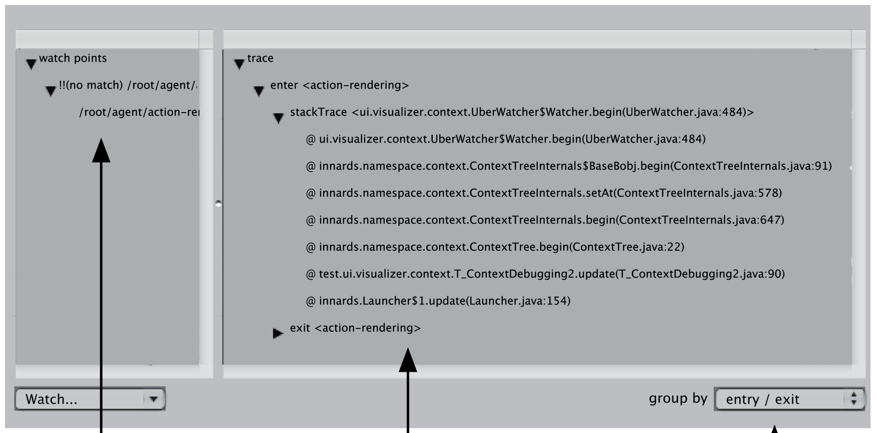
```
from marc.visualizer.context import *
class Gate(PythonGating):
        def set(self, context, key, old, new):
                #fill me in
                return 1
        def get(self, context, key, now):
                #fill me in
                return 1
ret = Gate()
```



```
▼ root
    graph_context0
  ▼ agent
    ▶ motor1-posegraph
    ▶ action-rendering
    ▶ misc-params
      storage-control
```

```
▼ context map
    cv_c[class innards.namespace.context.key.FKey  1.0
    cv_t[class innards.namespace.context.key.FKey  1.0
    inner_opacity[class innards.namespace.context  20.0
    line_opacity[class innards.namespace.context.k  0.0
    point_opacity[class innards.namespace.context  0.1
    point_size[class innards.namespace.context.ke  0.0
```

```
▼ watches
    /root/agent:point_opacity                          0.1
    /root/agent/action-rendering:inner_opacity         20.0
```



Watch...
group by
entry / exit
"breakpoints" set on context-tree access
full details, including stack-trace,of context-tree search
grouped by search context, find context or value

```
▼ watch points
  ▼ !!(no match) /root/agent/
      /root/agent/action-re
```

```
▼ trace
  ▼ enter <action-rendering>
    ▼ stackTrace <ui.visualizer.context.UberWatcher$Watcher.begin(UberWatcher.java:484)>
        @ ui.visualizer.context.UberWatcher$Watcher.begin(UberWatcher.java:484)
        @ innards.namespace.context.ContextTreeInternals$BaseBobj.begin(ContextTreeInternals.java:91)
        @ innards.namespace.context.ContextTreeInternals.setAt(ContextTreeInternals.java:578)
        @ innards.namespace.context.ContextTreeInternals.begin(ContextTreeInternals.java:647)
        @ innards.namespace.context.ContextTree.begin(ContextTree.java:22)
        @ test.ui.visualizer.context.T_ContextDebugging2.update(T_ContextDebugging2.java:90)
        @ innards.Launcher$1.update(Launcher.java:154)
    ▶ exit <action-rendering>
```

figure 60. The context tree becomes a place where we can focus the attention of custom debugging interfaces, to assist in the creation of agents. Shown here are the interfaces to the context-tree visualizer and the context-tree "breakpoint" interface. Breakpoints, which run on context and key access, are stored with the agent, and can be executed without any graphical intervention. Thus the boundary between "debugging" and "finished work" is blurred.

the interconnect between modules is traceable — the first step of all inter-module communication is a call to the context-tree interface. If we want code to monitor, reflect upon or perhaps even interpose itself between, the action system and M or the motor system and A, then the site of this intercession is clear. The result of this is that we focus a monitoring and tool building effort on the context tree, and for our tools and for our own navigations of the tree, we can exploit the hierarchy's implicit "locality of effect": changes (be they modifications or bugs) to a context affect only children.

the modules remain mobile — the motor system can be executed inside a different context (for example a different creature) and its binding lookups from the point of view of M will change to reflect the new context. For this to be useful, it should be easy to cache and store a state that needs to be context dependent as correctly context dependent. This is achieved by building higher-level container classes that are automatically backed by"context local storage" and by building language-level constructs that make, for example, "context locality" as easy to achieve as"instance locality" and"class locality" is in whatever object-oriented language that we have chosen, *page 204*.

This concludes the introduction to the context tree — my candidate replacement for an agent's central blackboard, an alternative to the arrows in a system diagram. The section that follows simply takes this structure and builds more useful structures on top of it. Compared to blackboard architecture, the context tree has more structure — a nest of searchable blackboards, an explicit and dynamic hierarchy of execution contexts that is tied to the execution of code. Compared to an"arrow" it has much less structure, a much less narrow focus; it provokes much weaker couplings between modules. Therefore the context tree, as proposed, is a more complex compromise between these two simple solutions. It sacrifices some of the apparent simplicities of either of these two positions, hopefully, in exchange for simplifying the real problems that come with their use.

3. The uses of the context tree

However, in our two examples above, none of these decoupling "victories" are secure against all possible manipulations of the contents, execution cycles and the assemblies of P, A, and M. And the real power that the context tree has over a central registry is that its internal structure somehow reflects automatically the execution and use patterns of the model that refer back to it. We'll need to see some more complex examples for this power to be obvious.

The Façade pattern is from: E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of reusable object-oriented software*. Addison-Wesley, 1995.

For example: what happens if there are *two* action systems to communicate with A_1 and A_2? Where does the second go? How does A_2 get a share of the communication between A_1 and M? Something similar to the well known Façade pattern might be deployed to make two action systems appear as one to the motor system, but just as the façade pattern hides its presence from the caller, it hides its presence from our previously traceable interconnect; it is a "local trick" and doesn't necessarily place control and share responsibility for its hidden manipulation in the right place. If we are going to perform such local maneuvers why have a central mechanism to begin with?

The actual problem behind this is an insufficient inversion of control — the motor system and a specific action system remain too strongly coupled together even when they find each other through looking each other up in the context tree. Better to note that much communication between the two systems can be articulated inside the context tree itself, with the posting and querying of results or elements that can be used to obtain results. Our pattern shifts then: rather than have a reference to *the-creature's-action-system* at the creature level (which is really the act to blame for our commitment to a single system at that level) we have our action system and motor system post results to and query information from the context tree — these results and queries are arbitrated by their own individual keys. Loose coupling can be achieved by using the context creature for this blackboard; results from both A_1 and A_2 are written here, in an execution dependent order.

Asymmetrically strong couplings can be achieved and with them a variety of execution independence:

A **stronger**, more detailed coupling, from A_x to M can be made by injecting a reference to A_x's context above M's context for use by M in looking up the results of A_x which are stored local to A_x — this coupling can be easily arranged by an assembly mechanism completely external to both A_x and M and independent of the ordering of A_1 and A_2;

A **local** coupling from A_x to M can be constructed by injecting the results of A directly into M's context — here A_x must come to know something about the existence of M's context, but nothing of the specifics of M's itself. Again this is (A_1,A_2) order independent; or

A **private** coupling by moving M to a sub-context of a particular A_x — this often makes A_x responsible for other aspects of M's life cycle including the ordering of A_x and M.

The context tree cannot make the decision as to which kind of coupling is more appropriate, but it does at least allow the decision to be made. And in each case, this serves the end of making the communication between modules more explicit, more standard (and thus observable by our generic context tree inspection tools) and, depending on how these results are described in code, potentially more declarative.

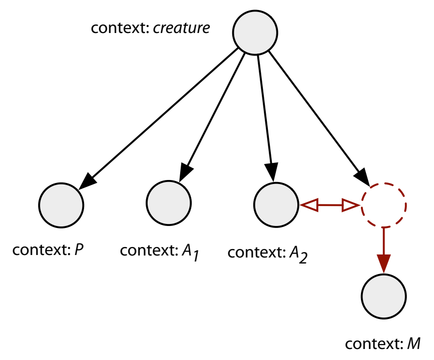


figure 61.A strong coupling between A_2 and M.

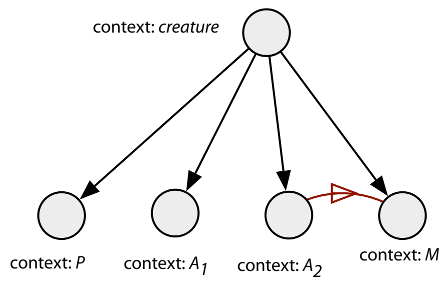


figure 62.A local coupling between A_2 and M.

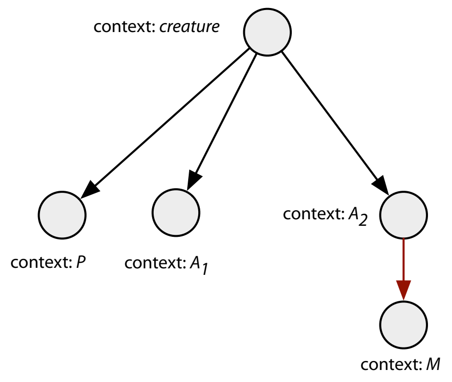


figure 63.A private coupling between A_2 and M.

Looking back at the complexities illustrated in chapter 2 on the "large-agent" system *alphaWolf*, we see a number that could be directly treated by using the context tree as a general mechanism for coupling modules, and it is not difficult to hypothesize uses for each of the above couplings in this project. The coupling diagrams of figure 15, *page 79*, tell three stories and hide three more. One is the sheer number of connections between disparate parts of code — which is in itself an argument for a strong and general purpose way of connecting things without boilerplate code. The second is the increase of connection density as the project grew. As modules "split" (presumably for the purposes of localization and testing), they duplicate and drag with them all of the previous connections to systems and add usually at least two more connections between the newly created modules — observe "fight action" and "toodle action" appearing out of the main "action system". Thirdly as the project develops, modules that were general purpose become specialized, and in doing so, other modules assume specific knowledge of that specialization — the transformation from "perception system" to "wolf perception system", and the leakage of this concrete implementation into other systems. Hidden here are the "modules" that are never created because the coupling to their environment would be so strong, when expressed in method calls and shared instance variables, as to render the exercise futile. Equally hidden, and equally absent, are the "mock" objects that were difficult to create for the purposes of testing other modules in the presence of such detailed shared knowledge of the implementation of each module. [ref mock object] Finally, hidden on the diagram, since it is difficult to represent pictorially, are the careful aggregation and ordering of one modules results by another in order to present this information in turn to other modules — for example, the perception system bundling information from the proprioception system in order to pass it to parts of the action system.

The context tree speaks to all of these problems. It offers a communicative complexity that scales with the number of modules, not the number of connections.

It offers the potential of the utterly"code-free" fission of a module since the ability to find and connect to modules is stored outside the module being fissioned. We shall see that the context-tree offers module-external methods of shielding the implementation details of a module from others; an more detailed, yet potentially safer coupling that decreases the the threshold for modularity; and, as illustrated above, a module-external place where code can aggregate and treat the output of several modules before passing them up or along the context-tree for the consumption of other modules.

However, in a more general sense, we have only begun the process of decoupling the modules of the agent — and we have simply deferred the problem rather than solved it if disparate parts of our hypothetical systems A or M need know about the specifics of the coupling arrangement — if special code need be written to implement these different classes of couplings. Rather, it should make no difference to the insides of the modules to access a variable in each of these coupling scenarios. It is to the decoupling of this decoupling that we turn next.

Context-tree container classes

Clearly we are presenting the idea that we can have a language-level binding that appears to be a something like a "regular variable" (for example, in java an instance member of some kind of reference type, in python a class attribute).

For example, in our most plain java we can define a context-tree key class with the following interface:

```
interface Key<t_value>{

     // looks up the binding for this key from the context tree
    t_value get();

     // sets this binding to be this value
    void set(t_value value);
}
```

This is the interface used for our previous examples.

In certain circumstances is possible to hide the presence of even this single level of indirection in Java, and in languages that are more directly malleable, such as Python, these meta-class level manipulations have been explicitly built in to the language. No matter, for even with this shim we can construct higher level data-structures where the references to objects have been replaced by these context-tree bindings.

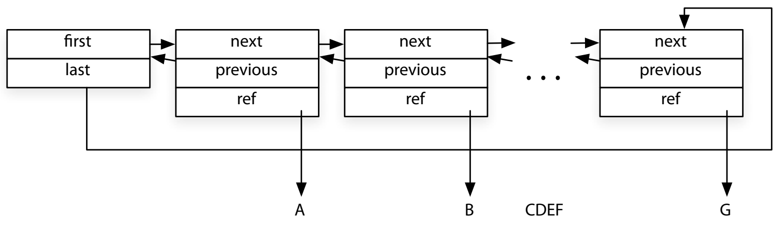
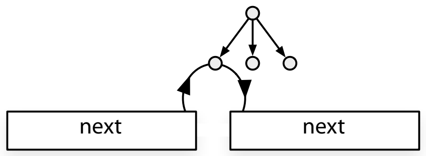


figure 64. The canonical linked list structure is an open network of references. We can replace the references with context-tree lookups to yield a context-tree-aware container class.



For example we can take the canonical linked-list structure and translate the *next, previous,* and *object* references (together with the *head* references) to unique context-tree bindings. This gives us a list container class that at each level of the context tree acts as a list just inherits the semantics of the hierarchical context tree lookup. In particular, changes to the list in parent contexts are visible in all children contexts while the opposite is never the case. This holds both for non-structural (changes of what a particular list element refers to) and structural alterations of the list (deletions and additions to the list); when structural modifications present at parents and children overlap, then the list appears to follow the principle that the child context overrules the parent context for the purposes of that child context.

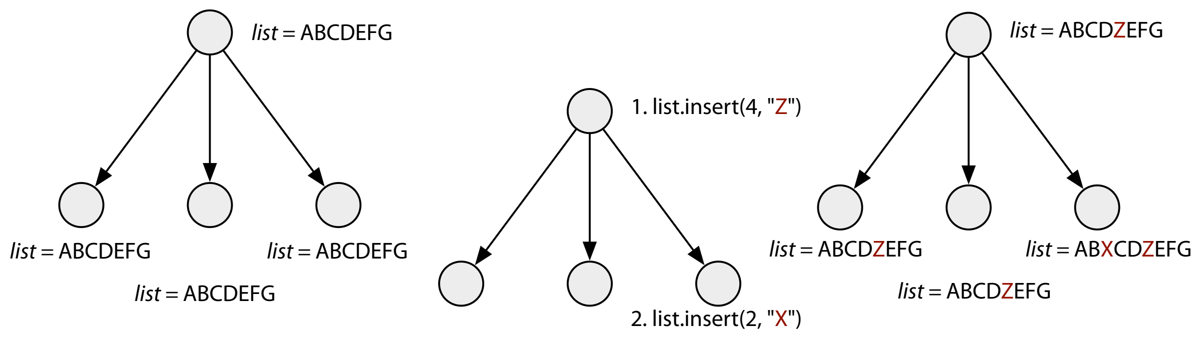


figure 65. The context-tree list extends the semantics of the context tree to a list container. Here elements stored in all parent contexts are visible in all children contexts. Similarly operations on the list, additions and deletions appear to occur at that level and all levels below.

The Java Collections framework is a standard toolkit of container classes — for a tutorial see:

http://java.sun.com/docs/books/tutorial/collections/

Similarly, "automatic" translation from instance-local to context-and-instance-local data-structures is trivial in the case of the (hash)map and the binary-tree, and thus is able to re-express the complete set of primary interfaces of the Java "collections framework". These context-tree-local container classes are then stored as conventional instance or class members, the further level of indirection afforded by storing these indirectly seldom being useful.

Programming in the interstices — code injection

These higher-level context storage containers allow the bootstrapping of yet higher-level programming techniques. When asked for its value, our context-key local class asks the context tree for the value of a binding and returns it. When used as a blackboard for the posting and retrieval of results, this is a site of communication between one module and another, and, as such, it is also a site of coupling of a different sort — one connected to systems' expectations of the semantics (what it means) of their communication rather than the syntax (where it is). We'll begin with a toy problem, although as we will see below this example occurred a great number of times during these multi-year projects, potentially threatening the very collaboration that was taking place.

A posts a value *A's result*:4.0 to the context tree and B looks for *A's result* as an important input. But the code for B was written a long, long time ago, and we've just compiled it again and mixed it into the agent in a hurry; A has been completely rewritten since then, and the scale of A's result has changed — how can we make it appear to B that A's units are twice as big?

One way to achieve the re-scaling is to execute code after A's posting but prior to B's reading that either rewrites the binding or injects a new binding into a context more local to B. These are clearly clumsy solutions, and neither of these solutions work well in practice. They introduce order-dependence where none previously existed and they interact poorly with a module C that is also interested in *A's result*.

A more interesting and maintainable technique is to open up the indirection provided by the context-tree key to the context tree itself.

Our key now looks like:

```
interface Key<t_value>{

    t_value get();
    void set(t_value value);

➡   void addToLookupStack(CodeElement<t_value> element);

}
```

where our CodeElement interface :

```
interface CodeElement<t_value>{

    t_value open(t_value filter);
    t_value close(t_value filter);
}
```

we can now restate key's **get** behavior in terms of this filtering stack, in Python-like pseudo-code:

```
returnValue = nothing;

for element in codeElements:
        returnValue = element.open(returnValue)

for element in reverse(codeElement):
        returnValue = element.close(returnValue)

return returnValue
```

Keys come with a CodeElement that heads their stack that just looks the key up from the context tree as usual. And of course, we can write a CodeElement that has an close() that performs the unit correction between B and A in a matter of a few lines. Maintaining two methods open() and close() allow filters to choose to pre-empt the main lookup. These CodeElement fragments are completely independent and potentially persistable.

The final twist is to make this stack of CodeElements a context-tree-local linked list. With this we can add our re-scaling code element to the list inside B's context and inside B's context alone. From within this context it is part of the stack executed to get at *A's result* and it gets its opportunity to modify the value passed through it accordingly; from outside this context it is not part of the list.

We are now in a position to begin to see the relationship between the context tree and inversion of control container systems. Comparing these context-tree extensible, context-tree lookup's to conventional IoC's push or pull"dependency injection" — the connection of instantiated modules — we might be tempted to claim the term"code injection" for the more aspect-oriented context-tree system. For here it isn't (just) the references that are getting hooked up through the context tree, but the semantics of the actual reference types offered by the underlying language that are being subtly stretched from outside the module of small amounts of code. This is neither strictly "pull" or "push" — these injections may come from other modules, typically parent modules, and one module's "central naming service" is another module's peer or child.

My context tree provides two other interstitial sites that are worth mentioning — *watches* and *traps*. Watches aid in the creation of context-tree debugging tools and allow code to be executed whenever slots in the context tree change. It is possible to implement this feature with zero additional cost in the case that there are no watches at or above a particular context, and this feature is designed mainly for debugging rather than self-monitoring. Traps, on the other hand, are called whenever contexts are entered, exited, re-parented, have children added to them, or deleted and are useful in maintaining caches of information (that may need to be updated if the topology of the context tree changes) or providing hooks for controlled shutdown in the case of context deletion. Later, I shall introduce structures that require this kind of caching in order to achieve good speeds at runtime, and we shall see some uses of the context tree for life cycle management in general, *page 215*.

Storing parts of the context tree — the technical support for naming

I've presented the context tree as a general purpose "working memory" for agents — a place where systems can post and read information and thus communicate in a loosely coupled fashion. This is an accurate description for its use in the agents in *The Music Creatures*, *Weather for an interactive window*, *Max*, *22*, *how long...* and *Imagery for Jeux Deux* and all other agents within the most recent versions of the c5 toolkit. However, the context tree began life as a solution to a slightly different problem — not as a substrate for communication between processes but as a place for communication between artist and agent during the creative process.

The hierarchical outline of the tree makes it ideal for configuring rendering parameters broadly, before stepping down a few levels of the tree for local control over a specific agent, a specific shape or a specific line. And being able to distribute code throughout the interstices of the rendering is of course extremely powerful — for example, this makes it easy to programmatically *distribute variety* — of rendering styles, or behavior, *page 98*.

This technique works so well that I invested much time in creating graphical and textural interfaces for the context tree to set, inspect and manipulate these values and injected code. The context tree, with its structure dynamically determined by the execution of the agent, is its own "ideal interface" for distributing parameters. Creating *Loops* and *The Music Creatures* was just as much a matter of traversing the context tree of the colony and manipulating parameters as it was of storing them, learning them and recalling them, *page 98*.

The filtration example above was presented as a toy example — and it certainly does look like a lot of trouble to go to just to multiply a number by two. However, this very problem appears frequently in practice. The context tree's scoped accessibility makes it an ideal place to put the large number of *ad hoc* parameters that get set and reset during the creation of an art work — be they line thicknesses, colors, noise parameters, filter coefficients. *how long...* moves around at least two hundred of these at various locations of the context tree, and almost everything that isn't specified in the action system in *Loops* is specified by these numbers — in the motor system, the graphics system and the global control of the colony.

However, there is a problem with storing these values. The resulting parameters aren't just numbers — they have been hard fought for and hard won; they might represent a considerable effort to tune the appearance of a line — e.g. *Loops* — there might have been a considerable amount of offline learning involved on the value of a coefficient — e.g. *Music Creatures* — they might represent a considerable amount of consensus inside a collaborative work as to what a particular appearance should be, last month or even last year — e.g. *how long....* Such is their importance in the development, one clearly needs a strategy for storing them past the life-cycle of a single execution of the work and past the memory span surrounding a rehearsal or an improvisation.

This storage of these parts of the context tree itself appears to be easily achievable by taking contexts and writing them to disk, and indeed this tree-like structure serializes to and from human-readable Xml extremely well. But these persistent numbers are not built from stable material, they are not referentially stable: line thicknesses are being developed at the same as the line drawer which today uses a selection of multiple lines rather than the simple one it did last week (*Loops*); a new agent wants to reuse the same rendering styles, on different geometry (*The Music Creatures*); learnt filter coefficients for one body now have to be translated into a new smoothing coefficients for a completely different body that as of last month hangs upside down — (*how long...*).

Where there is a *storage problem* there is a *versioning problem* waiting to happen: these numbers are not loaded back into the same system that saved them, and the effort devoted into finding these parameters cannot be allowed to slowly halt the further development of the system. If it could, we would be faced with choosing from two frustrating inertias — a forward inertia that would discourage us from changing the *process*, having made the *choice*; and a backward inertia that would discourage us from making the *choice*, until the *process* is "finalized". Both cripple the exploration of the field of potential developed by our complex systems just at the time when that potential might solidify into the particular; both threaten the collaborations that I have been involved in at, arguably its weakest place, my ability to regenerate material that was previously the subject of consensus early in the collaboration, despite the provisional nature of the systems generating it.

Having realized the significance of the problem, the solution turns on two ingredients added to our context-tree techniques. The first is a database that acts as a repository for named, subsets of keys take from branches of the context-tree — we'll call these subsets **persisted, partial trees**. These partial trees are stored with versioning information that exists, crucially, on two levels: at the level of the partial tree, and at the level of the individual key. Named partial trees are additionally arranged in this "database" into types: parameters for our line renderer would be a type; a variety of named partial trees, each with different names, would form a set of rendering styles that we are interested in deploying throughout the piece; and a set of partial trees of the same name might be an ordered history of how this particular rendering style has been modified during the development of the piece.

Particular dates can be identified with names inside the database — "before the January rehearsal" or "before line renderer 2 got fixed". Historical information about a particular key is never deleted, merely superseded; databases for the works presented in this thesis, some of which were constructed over a period of 2 years, reach a size of several megabytes. This size is perfectly manageable without recourse to more heavyweight, truly "database" back ends.

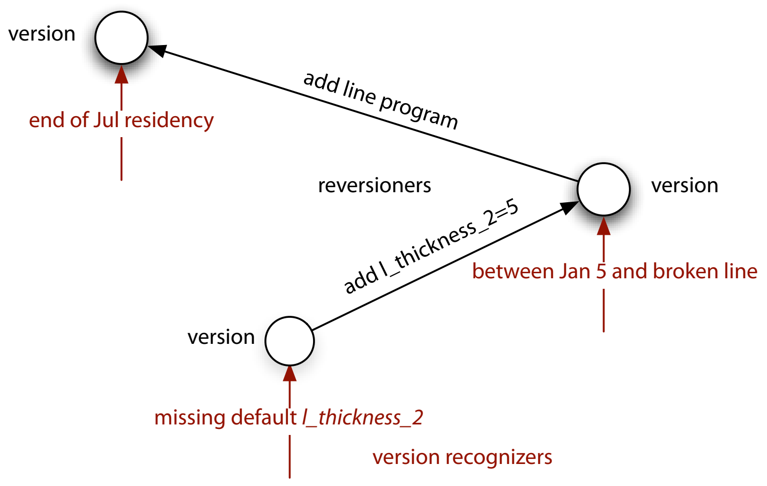


figure 66. Reversioners and version recognizers work together to ensure that persisted, partial trees are made up to date upon recall.

The second ingredient is a set of code resources that can first recognize when a key or a partial tree is "out of date" with respect to the current system and secondly do something about it. Version recognizers and "reversioners" can work in parallel and in series at both the partial tree and the individual key level. Neither recognizers nor reversioners can be made completely automatically, for the space of incompatible system changes resists standardization, but they can be made easily specifiable.

Version recognizers are generally quite simple, in most cases recognizing a specific date tag isn't the latest, or a particular key is missing from a persisted partial-tree. Reversioners generally act not by rewriting keys or adding previously missing keys but by injecting code into the keys or injecting code into new keys that tie their value to existing ones. The accumulated injected code allows old systems (or, more likely, old snippets of scripting code) talk the "old language" of the older keys while automatically presenting the newer interface to any module that cares.

Version recognizers are arranged and stored as vertices in a directed graph structure, with particular reversioners as edges of the graph, a path of accumulative "reversioning" is computed through directed search from the most recent (by date) version found from the database to the most recent (by date) version accessible to it in the graph. Should a reversioner $R_{1\rightarrow 2}$ that moves versions recognized by V_1 to those recognized by V_2 fail to turn a partial tree that is recognized as V_1 into a partial tree recognized by V_2 back-tracking occurs.

This "versioner graph" was successfully deployed in the pieces *how long…, 22, Lifelike* and in *The Music Creatures* and an earlier prototype of the system was developed for *Loops* and *Weather for an interactive window*. Three of these five pieces were collaborations, two took place over a period of two and a half years. The version graphs for each of these dance pieces contained on the order of tens of nodes, some general, but some quite specific. In *Loops* and sometimes in *The Music Creatures* these partial trees are the very material from which the lowest-level representation of the generic pose-graph motor systems deployed in that work — these named "rendering styles" were in fact the named "body configurations" of the agents of *Loops*.

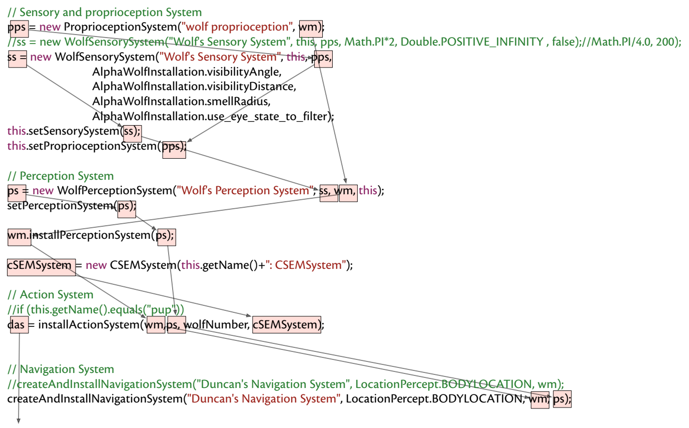
Indeed the small work, *Weather for an interactive window*, was mainly concerned with testing interfaces for live experimentation of rendering styles right up until minutes before the opening of the installation.

Although *Loops* exploited the context tree for its parameter distribution, it lacked a re-versioning graph system.

That the majority of these parameters were directly or indirectly related to rendering styles or body configurations probably stems from the the fact that these parameters are both the most readily placed as stored numbers or injected code and are the most likely site of tuning-while-running during a collaboration. However, it is also true that the results of offline and ongoing learning processes were stored and versioned in these persistence structures for maintenance across time-scales longer than the typical duration of the piece. Returning, once again, to *alphaWolf*, we can hypothesizes uses for these techniques not just in tuning, say, the rendering parameters for the wolfs, but rather in changing the way in which agents themselves were created. The partial storage of the context-tree, ought to have provided a mechanism by which particular arrangements of the social learning of the wolf-pups that were proving problematic to debug could be stored and recalled independent of the ongoing development of the social learning mechanisms. The ability to quickly return, in the presence of structural changes, to the debugging "frontier" would have greatly assisted the tuning of the social "game dynamics" of the piece.

Looking back to *The Music Creatures, page 127*, and forward to a more general-purpose expression of this idea, *page 390*, this is neither the first or the last persistence database that this thesis will discuss, and the theme of directly treating the historical development of a piece with custom tools continues: techniques that face up to responsibilities and consequences of long-term collaboration and thwart the encroachment of these choice / process inertias often developed when complex processes and artistic choice collide.

Authoring systems that change over time — the inverted context-tree list



```
// Sensory and proprioception System
pps = new ProprioceptionSystem("wolf proprioception", wm);
//ss = new WolfSensorySystem("Wolf's Sensory System", this, pps, Math.PI*2, Double.POSITIVE_INFINITY , false);//Math.PI/4.0, 200);
ss = new WolfSensorySystem("Wolf's Sensory System", this, pps,
                AlphaWolfInstallation.visibilityAngle,
                AlphaWolfInstallation.visibilityDistance,
                AlphaWolfInstallation.smellRadius,
                AlphaWolfInstallation.use_eye_state_to_filter);
this.setSensorySystem(ss);
this.setProprioceptionSystem(pps);

// Perception System
ps = new WolfPerceptionSystem("Wolf's Perception System", ss, wm, this);
setPerceptionSystem(ps);

wm.installPerceptionSystem(ps);

cSEMSystem = new CSEMSystem(this.getName()+": CSEMSystem");

// Action System
//if (this.getName().equals("pup"))
das = installActionSystem(wm, ps, wolfNumber, cSEMSystem);


// Navigation System
//createAndInstallNavigationSystem("Duncan's Navigation System", LocationPercept.BODYLOCATION, wm);
createAndInstallNavigationSystem("Duncan's Navigation System", LocationPercept.BODYLOCATION, wm, ps);
```

figure 67. A great many systems created and registered. The act of registering leaves a distributed, intricate trace that is hard to unravel.

The problems of constructing complex assemblages of interacting modules has motivated many of the interstitial techniques developed here. However, the problem of dismantling parts of these complex assemblages appears to be fundamentally even harder than constructing them in the first place. To see why this is the case, we should look back at the kinds of situations that appeared often in *alphaWolf*, where a great many systems were instantiated, registered and accreted.

Firstly we note that construction follows the execution of the code, but there is no closure around this or any construction in this object-oriented / imperative language. Each of those objects might wire themselves together and make more objects that keep references to others. This is easily written in an imperative style, and straightforward to execute assuming one has gotten the order correct, but the execution itself leaves no trace — it is possible and perhaps likely that these newly constructed, installed and registered objects don't have references to the objects that made them, installed them or maintain them as registered. Even if they do it's not clear that it is desirable for them to have the knowledge or the power to unmake, uninstall or de-register themselves. Such knowledge and access would represent a strong and, worse, diffuse and intricate coupling between disparate sub-parts.

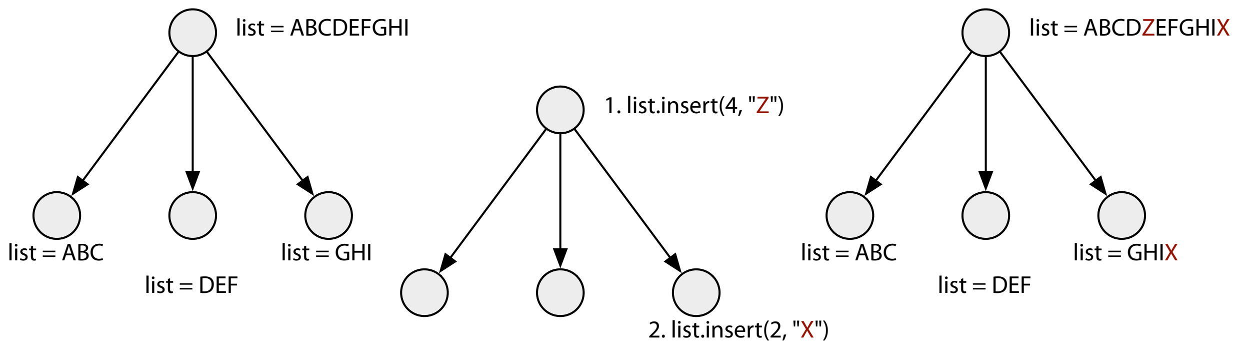


figure 68. The inverted context tree list appears to have the contents of a context and *all children* (as opposed to all parents).

Further complexity flows from the need to propagate a description of what is and what is not being torn down through the method calls doing the disassembly, it is not clear that this can this even be described without coupling the minutia of an assemblage to an external description of its boundaries. But we already have seen one description of the boundaries of a assemblage that doesn't need to be propagated anywhere — a branch of the context tree — and we have seen one species of container class that automatically reflects the state of the context tree. Perhaps these two ideas can be combined to make structural additions and deletions to the context tree equivalent to structural registrations and de-registrations.

Consider a new kind of list structure — the **inverted context-tree list**. Like our earlier linked list backed with context-local storage, its contents are actually distributed throughout the context tree, rather than stored in a particular instance or class field; and just like the our previous linked this this list is in actual fact a union of lists stored in a number of contexts. However, where the context-tree list was the union of all lists at the current context and its chain of parents, this inverted list is the union of all lists at the current context and all of its children. Unlike the previous linked list structure we cannot construct this list by replacing object-references with context-tree key lookups, for context-tree key lookups proceed, in the absence of code injection, upward through the tree. However, in practice we certainly can reuse the same caching mechanisms that make access to the context tree fast to maintain this inverted list of lists at each level of the tree.

This list is now a primitive from which we can construct registration lists, execution orders and notification queues. Whenever there is a list of objects that need updating or notified when events occur, this list should be an inverted context-tree list; whenever there is a map of known services that might be called upon by a module, this map should be an inverted context-tree map. Delete part of the context tree, and all references to objects installed at that level and below disappear; execute methods on this module in a different part of the context tree, and a different set of objects to update or services are available. Further, upon deletion, the references disappear in a particularly orderly and consistent fashion: namely, simultaneously. And this simultaneous destruction occurs at a particular moment: at the last transition out of the deleted context they are never seen again. This is a time when, by definition, there is no code running that accesses these data-structures. Together with a few "language level" programming tricks, this facility can be made broadly usable, removing much of the boilerplate code that is involved in both de-registering and even registering systems' connections, *page 231*.

This idea, as already described, is very useful beyond simply enabling the disassembly of systems — indeed it permits the rapid assembly of the kinds of complex assemblages that the context-tree promotes from going unexploited in systems that need to delete parts of themselves during their execution. Often in the face of creating an agent or an artwork, the focus is on assembling something, testing it, tuning it; having reached a point of confidence that that thing is heading the in right direction, one puts this piece down and takes up another part of the work. Only when these pieces come together do their life-cycles begin to get complicated: when we need to go from one piece to another, incorporate one in another, instantiate three things rather than one.

By no means, however, is this kind of tear-down essential to the creation of graphical, interactive agents — the installations *Loops, Dobie, alphaWolf, The Music Creatures, Lifelike* all ran without any structural deletion occurring during their life-cycles. Similarly, most interactive works — be they authored in Max, Isadora, Director, or Flash — seldom change structurally much after initialization time — data flows through pre-made networks of modules, pre-loaded resources are moved to the fore or hidden.

These environments will be the subject of much discussion in chapter 8 — see the references therein.

However, in each of my early works, the problems of structural deletion made their presence felt — *Loops* became an infinite piece about the finite materials from which it was constructed; *Dobie* learns without bound, without forgetting; *alphaWolf* faced difficulties of such magnitude loading and deleting its constituent wolves that after their five minute growth cycle they were swapped around rather than deleted and reinitialized. As I moved to agents with more complex parts, their lives, and the simulations they inhabit, grow shorter: *The Music Creatures* only lived for around 7 minutes before dying — and with their accumulation of models and graphical material they might not have made it much past a few hours if left to run; and *Lifelike* ran for the duration of a 30-minute dance work, but the accumulative flux of points, lines, graphical resources, and behavior systems was so great I fear it would not make it past an hour. Is this the necessary price for live structural change?

Instead, these works (and commercial interactive programming environments) find a solution space (or a duration) where they do not have to confront the issues that come with tearing down previously constructed systems that may have changed structurally; but at the same time, we are clearly interested in works that do change structurally and remain running over long time-scales.

So the above techniques, the context tree and the containers, enable *how long...* to have a parade of overlapping, interactive agents that come and go, that model and stop modeling, that add and subtract geometry. As an artwork there is no secret reason why it could not run indefinitely. If it were appropriate, we could loosen the scripting of the entrances and exits of agents to create an endless chance juxtapositions of bodies and perception systems. Perhaps this flexibility will be demonstrated an installation context at some future point. Regardless, the flexibility paid for itself in rehearsal.

Therefore, together the context tree and its container classes allow for the assembly and automatic disassembly of modules. However, there is an alternative interpretation of this power that has implications squarely in the domain of artificial intelligence, not simply software engineering. Since the addition and deletion of things are tied to the context tree and since contexts nest we can use context-trees to implement structural *closures*. That is, we can open a child context, try some computation and, if we don't like the results just delete the context and it is as if nothing happened. If we do like the results, we need to propagate the contents of that child context up to the parent — overwriting slots that contain simple values, merging slots that contain lists etc. This kind of **speculative execution** allows systems to hypothesize about what would happen if it changed in a certain way. Unlike languages with built-in support for closures it is up to the programmer to decide what is and what isn't closed in. This is a problem since it is prone to error unless using context-tree-local storage is simple (see the annotation library, *page 226*), but it is a benefit because it allows us to choose what does and what doesn't get rolled back on this context-tree-level "undo". This technique has a number of in the works: *The Music Creatures* used an early version of this technique to speculatively close a sensitive period that might in fact need to open again; *Loops Score* uses it in a number of places to see if performing a musical action will produce a series of notes that can be related in some way to what has already been scheduled, *page 235*. I expect that this technique will find an increasing role in future agents that undergo long execution, long-term structural change.

Further, such a facility — had it been offered by the toolkit during at that time — would have radically changed how projects such as *alphaWolf* were authored. Not only would it have been possible to unravel the kinds of structures indicated in figure 67, but this error-prone glue code itself would disappear. By preserving the modularity of parts of the agents one could also prevent the unfortunate collapse of all three "kinds" of agents (pup, adult and caretaker) into a single action system — allowing the set of action tuples present in the system to grow and change during the life-cycle of the creature rather than creating a single action system with parts "shorted-out". This offers an improvement not just of computational complexity of running the action systems (which, in itself is dwarfed by the graphics and animation tasks) but of the complexity of debugging these systems, visualizing their execution, even just thinking-through the results of modifying the source code (see, for example, the number of "optional sections" in the *alphaWolf* behavior code in figure 69, *page 222*). Further, a real deployment of the context-tree as a universal coupling between parts of an *alphaWolf* creature may have permitted modular testing, in particular the profiling and creation of the perception system and motor system independent of the action system, allowing in turn the multiple programming collaborators on that project to work more independently (directly treating the problems illustrated in figure 15, *page 79*).

The context-tree and system creation — sub-classable complex systems

The above sections have considered the context-tree-based solutions to the problems of constructing, manipulating, and disassembling complex assemblages of interacting "modules". This last section addresses the related problem of re-using them.

Often in object-oriented programming languages one expects to be able to use inheritance to provide a set of good base classes that will speed the implementation of, and reduce the amount of code required for, common specific classes. In an agent toolkit one expects to be able to create a simple agent in code by perhaps overriding a few methods from a base class that aggregates the machinery required for a graphical agent, an agent with a motor system, a perception system and an action system etc. Unfortunately, in practice, building a toolkit that offered such a super-class template has proved to be extremely difficult, and throughout the collaborative development of the 'c' series of agent toolkits, there has been an ongoing tension between powerful "abstractions" and useful "base-classes" . Here a apparently irreconcilable tension appears: between concretizing all-too-abstract elements in order to make specific agents, or abstracting all-too-concrete previous works to make the next. *AlphaWolf* falls, in my opinion, squarely inside the hole between these two poles — little of the code piece gets reused in the works that follow, yet the generic nature of some of the elements used in its creation can do little to prevent the growth of the behavior system files.

In this quandary, nothing less than one's technical development as an artist is at stake — the tension is between maintaining a broad ranging set of elements that are difficult to turn into free-standing artworks, versus accumulating knowledge and experience, but no tangible tools or code.

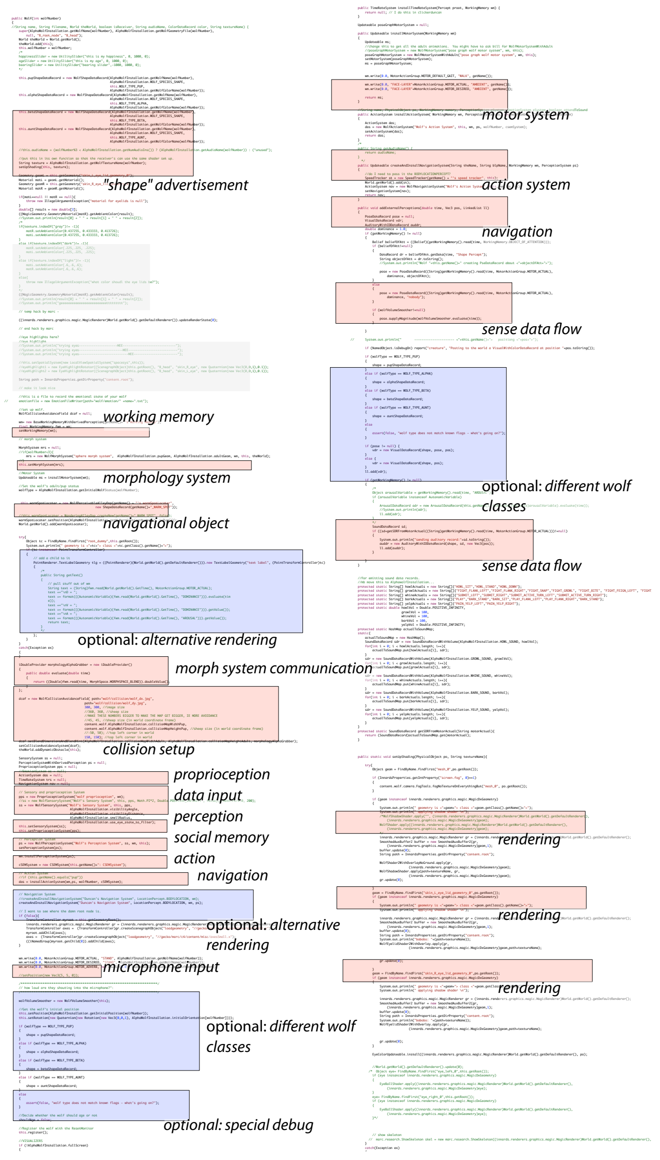


figure 69. Optional blocks (blue) and class instantiations (red) are distributed broadly through the *alphaWolf* action system.

Let us look at this complex "sub-classing" problem in detail. Perhaps the problem stems from the number of *options* that this super-class aggregation needs to present to its possible sub-classes. Rather than reducing possibilities as we descend down the inheritance chain to get closer to specific uses of the agent toolkit, the opposite happens: flexibility increases exponentially as we aggregate systems that are suffering from the same problem. So, unless there exists a mechanism for distributing defaults and overrides amongst these systems, either inopportune choices are made and the inheritance hierarchy is abandoned, or the base-classes provide so broad a base that they are hard to understand.

So far we have tried hard to allow the creation of complex assemblages that do not couple to their connections — essentially, their "parameters". The context tree defuses some of these tensions. We can have constructors for these aggregating classes that are written in a normal fashion — passing through, incorporating and storing parameters — as well as mutators that are coded almost as normal; all using the context tree. However, these classes do still couple in a number of ways to the classes that they instantiate. This instantiation is one way that super-classes commit to limiting the options for their potential future sub-classes and if this instantiation cannot be defaulted and overridden then we are very much in the situation outlined above.

The standard design pattern used to avoid this in practice is another inversion-of-control technique: the well known factory pattern. The factory pattern interposes logic in the name resolution and construction of classes. Rather than asking the language for a new instance of a specific class, one asks the factory for a new instance of a particular interface. This factory *decides* on what class should be instantiated and with what *parameters*. There are two problems associated with trying to deploy the factory pattern widely: providing enough information to the factory for that decision to take place and providing those parameters once that decision has occurred.

The context tree can clearly help with the distribution of parameters across a set of systems — this is the very purpose to which we have put it in the examples above. This ability could be pressed into backing a factory system — we could distribute the names of classes to instantiate just as easily as we can distribute other parameters. But perhaps there is a more apt and more flexible meeting of the context tree and the factory pattern.

We can use a version of the interstitial context-tree programming technique described above, *page 206*. Instead of shielding systems from changes that are incompatible with previously saved data, by installing context-specific filters on access to that data, we can install context-specific factory functions to instantiate classes.

We build up the language very similarly — and we should build it up, for it is going to become a common *programming* unit for a whole variety of systems. They key programming element is the the CodeElement from before:

```
interface CodeElement<t_value>{

        t_value open(t_value filter);
        t_value close(t_value filter);
}
```

These elements are stored and executed in context-local lists inside ExtendedKeys. This class offers what Key offers plus a number of convenient versions of the add method that help write more factory-based context tree programs. The most important of these methods are:

A little care must be taken with how Java's generics and type system are used to make this syntax work. In general we admit not only t_value, the underlying value type for the keys, but <? extends t_value>, and Class<? extends t_value> to be passed into these methods. These methods are omitted for brevity. Internally, of course, Keys are free to ignore the parameter on the type, since the implementation of generics is not strictly part of Java's type system.

call(Object *o*) — wraps any object (that is presumed to have only one publicly accessible method) in a CodeElement and adds call to this element on the stack.

lookup(Key<t_value> *o*) — looks up a key from the context tree.

is(t_value *o*) — just returns this value into the execution order of the stack.

with(Key<t_value> *key*, t_value *value*) — places a CodeElement that temporarily sets a particular key to a particular value on the stack.

makeNew() — puts a CodeElement on the stack that will instantiate classes that are passed to it (runs in close(...))

This lets us declare a default instantiation, perhaps in a creature base-class, like:

```
navigationSystemFactory.is(DefaultNavigation.class).makeNew();
```

And then from some other location, perhaps in the action system, where we need a navigation system:

```
navigationSystemFactory.get();
```

will suffice. This is the most straightforward of all possible examples. We can use the context tree to pass "keyword parameters" to this factory:

```
navigationSystemFactory.with(bodySize, 10).get();
```

We can, far away from this invocation or declaration, manipulate the factory lookup such that this action system gets a special navigation system, perhaps in a subclass of the base creature class:

```
navigationSystemFactory.inside("action-sys/").is(GraphicalNavigationSystem.class);
```

Or, less radically, we could just provide a better default:

```
navigationSystemFactory.with(bodySize, myBodySize);
```

XPath is a w3c spectfication — http://www.w3.org/TR/xpath

The flexible Java XPath engine used for this work is Jaxen — http://jaxen.org/faq.html

This project specifically allows XPath expressions to be evaluated over "custom" object models (here, the context tree, later, the *Fluid* view hierarchy).

In order to distribute these throughout the partially assembled context tree it is useful to have a path expression language to refer to contexts different from the current one. Rather than invent an expression language that handles hierarchies of attributed objects we borrow an extremely well thought-out, standardized expression language made for searching xml —XPath — and map its object model (which is usually the tree-like structure of elements and attributes in xml) onto the context tree (a structure of contexts and keys). This allows simple searching for contexts by name over the whole tree (looking for "action-sys" regardless of where we are):

```
navigationSystemFactory.inside("//action-sys/")...
```

as well as more complex expressions that may match multiple contexts (looking for a context that contains a value for *direction*):

```
navigationSystemFactory.insideAll("//*[ct:containsKey('direction')]")...
```

The methods that round out this interstitial programming environment are:

and(Object *o*) — call-s *o* should nothing have been found yet.

beforeAnd(Object *o*) — call-s *o* first in the stack, and only calls the rest of the stack if it doesn't come up with an object.

importing(Object *o*) — imports the context given by *o* into the stack at this point.

There is a strong similarity between these methods and the complex, multiple dispatch of the Common Lisp Object system. However, here, dispatch is extended in a context-centric way *external* to the class or method structure target. This borrows, then, the flavor of Aspect Orientation but is specifically focused on the problem of instantiation and allows configuration behavior to be modified live, on a per-context, that is a per-hierarchically-defined-module, basis reusing the functional groups that the context tree represents.

While these techniques do not make the problems of extending complex code assemblages evaporate they do make a great deal of different to the problem. Unlike the tangled "excesses" of earlier agents that, while informing the work that followed, were, as a set of code, ultimately abandoned after premiering, the agents of *how long...* were constructed both with more generic element and more simply. Thus, we will see these techniques exploited in the agents of *how long...* — in the LineAcceptor system, *page 306*, and the "subclasses" of the agents built between workshops and in the evenings between rehearsals would not have been possible without this mobility. Intense, but open collaborative practice requires a solution to the abstract and flexible / concrete and useful dichotomy — one's prior work to entering the theater must be useful if there's work to be done, but it must be flexible if there's a collaboration that is to occur.

3. An annotation tag library for context-tree use in Java

Although it is certainly possible to implement the above context-tree key class, and use the context tree itself in pure Java, the works discussed in this thesis have gravitated towards one of two alternative paths. They either provided extended, syntactic "sugar" for the context tree in the form of a pre-processor language for Java or, more recently, a set of method and member annotations and a custom, byte-code injecting classloader that manipulates classes based on these annotations.

This latter implementation, which has only become possible with the most recent revision of the Java language, has the considerable benefit of standardization, allowing one to maintain an utterly conventional tool chain in the presence of even radical alterations to the semantics of Methods and Classes. Because of this, it will be this implementation that will be described here. It is in this "tag library" that we are the closest to "Aspect Oriented Programing", but mainstream AoP lacks the idea of a context-tree.

Java's annotations (similar in implementation to the annotations of C#) are essentially programmer-defined, structured "comments" in code that can be read at run-time, or in this case, class-load time. We start with our most basic annotation that tags classes:

@context

this informs our custom class-loader that this class has the potential to have the other tags from our library. It is a class-load-time error for subclasses of such classes to omit this tag and debug-time error for other tags to be present in a class without this tag. It provides both safety and optimization for the load time.

@dynamic(*name*=optional prefix)

marks this member variable as being context-local. subsequent Write and read access to and from this variable is rewritten to go through a context key. This context key's identifier defaults to the name of the member + the name of the class. Currently, only private object reference members of classes may have this tag.

@subcontext(*name*)

places the contents of this method inside a child of the current context called "name", creating this context if necessary. Crucially this has three properties that are hard to get right without language support: the tag exhibits the expected behavior for overridden methods, specifically their method bodies are wrapped and wrapped once regardless of the existence, location and number of any calls to methods in the superclass; secondly the context is correctly unwound should the method exit abnormally; and lastly this tag has the expected behavior even when the method annotated is a constructor.

@inside(*contextdescription*, *creationdescription*=default)

a more flexible form of "subcontext" that allows specification of two helper classes that define how to find the subcontext and how to create it should it not be found. In addition to the helper classes that "subcontext" uses that finds a named child context of the current context and creates one if it isn't there, other helpers have been found to be useful. One acts as "subcontext" does once and then from that point on goes back to that exact same context, regardless of the current context. The current context is restored upon the exit of the method. This is useful, for example, to ensure that methods of an instance are executed in the same context that was present when the instance was constructed.

@deferUntilEntry(*contextdescription, queuedescription*=default)

@deferUntilExit(*contextdescription, queuedescription*=default)

these two tags **defer the execution** of the body of the method until a specific context is entered or exited. These tags are only to be used on methods that have no return value. The queue description describes a class responsible for storing the deferred method calls, which need not be a simple list: subclasses that concatenate multiple method calls together into a single call are useful, more complex deferrals will be discussed, *page 258*.

These annotations clearly give a lot of power to a class to alter the scope surrounding the execution of a method in a way that maintains a coupling between object and execution context. They are not arbitrarily powerful — for example, it is impossible with these tags (but not with the underlying interfaces) to change the context in which a method executes based on a parameter to that method.

@autoUpdates

this tag marks a constructor of a class (or a whole class, thus marking all constructors). Constructors marked this way register the constructed instance with a **central auto-update list** fetched from context-tree local storage which, by default is an inverted context-tree list, *216*. Instances constructed this way will have their update(...) method called when the auto-update chain is updated. The interface for Updateables is trivial:

```
interface Updateable{
    void update();
}
```

A more complex life-cycle interface is optionally:

```
interface Task extends Updateable {
    void init();
    void update();
    void shutdown();
    void forceShutdown();
    void isShuttingDown();
    void hasShutdown();
}
```

This fuses two models together, optional shutdown and forced shutdown. A life-cycle object that has shutdown() called on it may opt to return *true* from isShuttingDown() and may ultimately return *true* from hasShutdown(), guaranteeing that this object will never have update() called on it again from this context. Alternatively, should the updator call forceShutdown() this component must expect never to have update called on it again (again, from this context). In this case, components that absolutely must shut down over a period of a few calls to update() should arrange for their update() to be called by some other means.

These life-cycle interfaces are implemented by a wide range of classes throughout *how long...*, 22, and *Loops Score*; the agents of *how long...* themselves implement these interfaces, as well as the rendering tasks that require graphics resources, and the individual graphical elements and scripts in the *Fluid* environment.

The deletion of contexts, which would silently prevent subsequent calls to update() for autoUpdatable-s stored in inverted context-tree lists without any call to shutdown(), and therefore violate the implied contract are monitored for by installing"traps" at the current level of the context tree.

@doesAutoUpdating
@doesAutoUpdatingOverContext
@doAutoUpdate

These tags are for classes that do the updating — instances that will be containers for other classes. The first tag marks a constructor of a class (or a whole class, thus marking all constructors) as owning part of the context-tree local auto update tree. The constructor (including super constructor invocations, and all inherited subclass constructors) is effectively wrapped in a pair of save(...) and restore(...) calls for the context-tree local storage for the central auto-update list. The second tag uses a inverted context-tree list rather than a conventional list to back the storage of this local update loop. This is useful in the case where a single instance is expecting to be updated in several times in different contexts. The third tag performs this auto-updating in the body of the following method in a "overriding safe" manner.

Of course, a majority of classes that are marked @doesAutoUpdating are marked @autoUpdates as well — they are "updatable" containers that create things that are updated in turn.

Several more tags will be added to this library through the same mechanism to complete the Diagram system, they integrate some of the features of that work back into the language — transforming what looks to the caller to be a method call into an object that is deferred, executable, inspectable, modifiable.

The Diagram framework — the channel / marker representation

In *Francis Bacon: the logic of sensation*, French philosopher Gilles Deleuze presents the diagram as a stage of artistic creation, citing Bacon's example of a brush stroke which reveals that the mouth of a portrait could cut across the entire face, suddenly increasing the sense of distance and transforming the figuration. Deleuze affirms the role of chance in this act — the lines of the diagram are "irrational, involuntary, accidental, free, and haphazard". The diagram exists on the boundary between preparatory work and the act of painting proper; its chaos must be transformed into a new form of figuration.

G. Deleuze, *Francis Bacon: The Logic of Sensation*, D.W. Smith , T. Conley (trans.), University of Minnesota Press, 2003.

The Diagram framework, in this work, synthesizes many of the technologies described above to generate a canvas on which unsynchronized or uncoupled processes, perhaps action-selection systems or perceptual traces, can mark — and out of which a new figuration (here a new temporal patterning) can be found. It is simultaneously a loosening of techniques like the c5 action-group, hierarchical scene-graph-based graphics and the pose-graph motor system, and an opportunity for more direct temporal specification.

There are two extremely minimal core elements to the Diagram framework: the **channel** and the **marker**. A marker is an object with a time and duration with respect to some time-base. A channel is an set of markers, ordered by onset time that shares a time-base with its markers. A channel has a unique channel context — that is, a part of a *local* context-tree that is associated with the Diagram system. All diagram system storage is local with respect to this context tree, thus all marker creation, modification and deletion is potentially speculative. Markers are always part of one and only one channel.

This micro-structure is generic enough to permit the re-expression of many of the data structures seen thus far. This channel / marker structure is a minimal subset of the representation behind the generic radial-basis channel, and behind the visual elements laid out in a sheet in Fluid, it will become our instantiated action, and our scheduled pose in the pose-graph motor system.

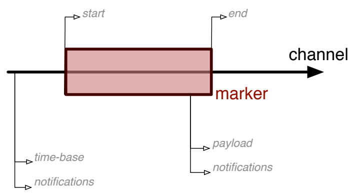


figure 70. The basic anatomy of a marker and a channel.

The first layer of Diagram focuses on building channels and markers extremely well. In particular it is occupied by building notification mechanisms (for the addition, change and deletion of markers) with respect to whole channels or individual markers. These notification chains support batching and event coalescing, *page* 258, and form the basis of the efficiency and efficacy of the algorithms that use these channels. Both channels and markers are mutable, but can provide immutable views as well as views onto windows of time or segments.

A detailed discussion of these inner details here would be unmotivated, so we will wait until the uses of the framework come into focus in making *Loops* Score before describing the implementation aspects. The goal for the Diagram system is not to provide another action selection algorithm (although we shall see one, *page 235*), nor another perceptual framework (although, *page 242*), nor a replacement for the pose-graph motor system (although, by the time we see the *Fluid* environment Diagram will have turned the pose-graph inside out, *page 329*). Part of the goal is to simply re-articulate the previous structures in a form that reduces the barrier for interaction between them. But in doing so I will present a new "recomposition" of action and motor patterning. The goal is to build a support structure around these algorithms that extends their power, specifically in the realm of supporting complex temporal patterning, specifically for the use of the agent metaphor in time-based art. It renders explicit, indeed *graphic*, what was previously implicitly hidden in action system initialization code, the trace of execution through a motor system or the pattern of activation in a perception system. It allows the re-coupling of algorithmic processes that cut across systems and algorithms in the temporal domain.

Some principles, however, guide the development of the Diagram-based works and the uses to which we put the channels and markers — *Loops Score*, *how long...*, and to a lesser extent *22*, *Imagery for Jeux Deux* and *The Music Creatures* (which developed an older version of this Diagram work).

These principles are:

marker manipulation should be **reversible and inspectable** — we shall see algorithms for marker production (essentially nothing more than production systems implemented with an explicit time axis) and manipulation,

The c5 + Diagram Action Selection Algorithm

state (all state marked @dynamic)

startles — a set of action triggers that get special privilege to interrupt others.

triggers — a set of action triggers inside this group.

lastValues — a mapping from tuple to real number

budget — an object that will handle the balancing of the action budget and hold the "list" of current actions

algorithm

```
if the maximum tuple.expectedValueOf() over all of startles is greater than zero then
nextOffer becomes the
greatest of these.
▸ otherwise,
    construct the new map nextValues[trigger] = trigger.expectedValueOf() for all triggers
    if any trigger has nextValue[trigger]>lastValues[trigger] and
    nextValue[trigger] > 2* min(budget.currentlyActive()) then select a new action
    if budget.isUnbalenced() then select a new action
    ▸ if we need to select a new action:
        if all of nextValues[...] = 0 then nothing is done
        ▸ otherwise,
            sample nextOffer from a normalized version of nextValues[...]
            success = budget.offer(nextOffer)
            ▸ if success:
                nextOffer.instantitate()
                and set lastValues[...] = nextValues[...]
```

but very little information is ever irreversibly removed or irretrievably hidden in any of these manipulations. Rather than translating a marker by overwriting its position, a relationship is set up and maintained that actively moves a marker to the left. This relationship is added to the channel, visible in the diagram, inspectable by other processes. Where action-selection systems allow code to compete for expression, the Diagram system additionally allows processes to compete for the modification of expressed lines or channels. Notification and compression support make reading and storing (and strategically forgetting relationships) computationally tenable.

transient, experimental computation — on the other hand, the computational impact of this agglomerative network of relationships is bounded by the transient nature of these channels. Often they are only representing the near future, and at one end there is the present — a scheduler horizon — and at the other there is an increasing uncertainty about the future — a planning horizon. The detailed relationships are meant to be transient, compressed and discarded. Slicing channels and modifiable sub-views onto channels help with writing code that efficiently deals with windowed portions of time. Context-Tree backed storage allows structural modification to channels to be attempted in sub-context, experimentally conducted only to be effortlessly discarded.

highly accessible — the channel / marker idea would be abstract beyond the point of utility if it wasn't for the common set of "glue systems" that make Diagram-like computations fast and allows them all to share a common language. The idea itself is useless, and in addition not very diagram-like, should these channels and markers be inaccessible to systems that have little commitment to the Diagram system. In the work that follows the channel / marker system is forced into systems by extending the programming language, *page 226*, aggressively finding commonalities between it and the visual tools (*Fluid, page 393*), between it and more traditional computer musical concerns (*Loops Score*) and between it and the way that motor systems are structured (*Loops Score, page 250*). Diagram will blur the separation between action selection (or more strictly action *layout*), the kind of motor sequencing that agents tend to do and the previous working memory / context-tree "blackboard" that was used as a communicative glue between systems.

As the description of the work that is based on this kernel continues it should be more apparent just what it is that is diagram-like about this diagram framework. The Diagram system's channels become a field, a rapidly receding canvas where multiple systems leave initially uncoordinated marks in time only to have their chance-like relationships enforced, reorganized or reshaped by other processes that induce patterns out of them and their histories.

Action selection in the Diagram framework

Diagram, as described up until this point, is missing a key part of the action system story — an action-selection strategy. While we would be free to take a c5 implementation of action-tuples and have its actions mark channels, it should be clear that this algorithm isn't quite taking full advantage of the channel / marker representation. Firstly, this approach knows little and says less about any time other than the present. Secondly, it doesn't then open up its action selection to external modification — its action-selection results are neither reversible or inspectable. Thirdly, at the core of c5, as it is typically deployed, is an assumption that there is only one, always one and exactly one active action at any time. Of course, creatures constructed with this approach are free to have multiple c5 action groups operating in parallel — and we have seen such creatures in *alphaWolf* and even the colony as a whole in *Loops* — but in Diagram we are particularly interested in this problem of multiple action selection in order for these multiple actions to be coordinated. Specifically, it isn't clear that delegating multiple action selection to multiple independent and continually overlapping groups isn't simply deferring a problem of coordination rather than solving it.

My approach is to disassemble the c43/c5 + action-tuple organization and then reassemble it in a slightly different order, ultimately exposing its internals as a diagram-like representation.

While the action-tuple is a convenient shape to draw and a convenient chunk to think about we will have to split it into two to prepare it for multiple simultaneous actions (additionally, in the overview of c43/c5 we already saw that an action-tuple had to expose its .trigger() .value() and .doWhile() methods separately. The unity of the action-tuple is clearly already under attack).

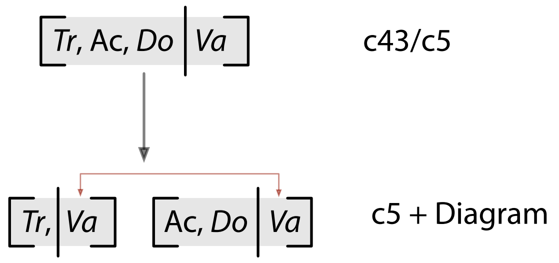


figure 71. The action-tuple is split up into two parts, a trigger factory and an produced action.

The *trigger* becomes, in addition to the place where we obtain the instantaneous relevance and expected value for the action, the factory for the *action payload* — an object that, when asked, is capable of producing an action which then exists independently of the *trigger*. This formalizes the split described above— although many factories and payloads collaborate after creation — and it allows triggers to effectively "group" parametric actions together and possibly instantiate multiple versions of them simultaneously, or even just one after another.

The second modification to c43/c5 is to explicitly allow multiple simultaneous actions — assuming the trigger factories are willing to allow it. We define another object that represents the *action budget* — the number of simultaneous actions that this action group is willing to maintain. With the help of this object, the core c43/c5 proceeds as normal — the central heuristic of only selecting actions when the action-group's triggers and do-whiles have changed both significantly and relevantly remains, only here an action selection may result in an action-tuple being added to the active set rather than replacing it, or the outgoing, loosing action being deleted from the active set according to the demands of the action budget.

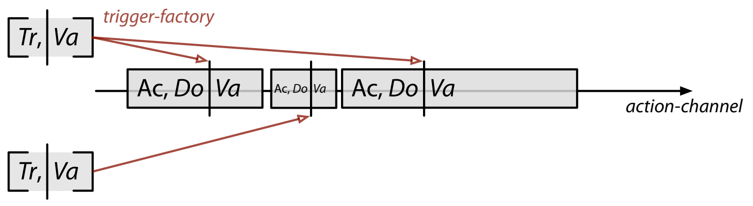


figure 72. The trigger factories schedule produced actions into a channel which is then read out over time to execute the actions.

The final twist is to both store and present the results of the action-selection strategy in a publicly accessible channel. Running actions are markers, triggers produce markers when they are selected for. Indeed triggers do not simply instantiate actions, they **schedule** actions by writing into the channel, and are free to place the results of their winning selection in the future. All storage concerning whether an action is or is not active — in particular, for the purposes of maintaining this "budget" of multiple actions — is maintained with respect to this channel, and a number of other, auxiliary and independent processes can act on this channel without fear of disturbing the action-selection process. These processes fine-tune, realign, filter, recombine and even delete the scheduled actions.

Many of the subtle timing and organization issues of the c43/c5 selection algorithm either evaporate or become explicit and visualizable rather than implicit. Through the trigger / action split we have made explicit the possibilities for actions to participate in more than one group and now have a focused location for more complex cross inhibition. Through the open channel / marker representation, temporal coordinations that would have to be implicitly coaxed out of the temporal dynamics of the triggers and do-whiles can now be specified as quite self-contained processes orthogonal to action selection itself — and we shall see many examples of processes that cut across the results of action selection, clean them up and constrain their relationships.

These post-selection modifications need not be kept strictly downstream of the trigger instantiations — by pushing all of the action-selection and budget storage into the context-tree we can even allow for speculatively executed, *page 215*, action selection, and more usefully, action instantiation. An example: triggers can retract their instantiated actions, "unscheduling" them from the channel. They might do this because, after the application of the processes that would clean up the scheduling of actions in the channel, some condition is not true. The only signal propagation out of this speculative execution"closure" is the the trace that the *trigger* should not attempt to re-fire.

Finally, by creating a persistent trace of the action execution, we formalize some of the ad hoc nature of the deferred credit assignment that we found necessary for trainable characters, *page 71* (a direct use of this flexibility, *page 324*); one can imagine other processes scavenging inputs from the history of execution — the channel of markers past.

Although it should be clear that, at least in principle, the availability of post-selection, *ad hoc* filtering processes increases the both expressive range of action-selection within the c5 toolkit and the vocabulary available to the agent author for articulating that range, it is worth pausing to review what such an extended action selection algorithm might have meant for the *alphaWolf* project. The Diagram framework explicitly treats many of the problems that caused profound complexity in the creation of both the action systems of the wolf-pups and its interface to the motor systems. Firstly it is important to realize that a pup action system is in fact an example of a multiple, parallel action system — each consists of an"attention" action group and a"main" action group. However, while the execution number and order of these units are fixed by the limits of the toolkit at that time (the attention group always runs first, and both groups only select once), the order that makes sense to the problem that the behavior designer is trying to solve changes depending on the part of the interaction being authored. Sometimes what the pup should pay attention to is limited by the action that the wolf pup is performing (e.g. it should look at the thing that its fighting); at other times the actions that ought to be performed are constrained by the object of attention (e.g. the pup can't fight a navigational marker on the ground supplied by the person directing the pup). What raises the stakes of the problem is the paucity of ways in which these two groups can be coupled — the first group to happen to make a decision gets priority and one ends up fighting against the protection against dithering inherent in c5 — the mechanisms to bias the rolls of the dice of these action groups are abused to ensure that the right set of dice gets rolled first. This is the spindle around which the complexity of figure 16, *page 80*, weaves itself.

In Diagram the coupling is much more easily expressed as a filtration process on the results of a single, much simpler action group. Weights or biases on the triggers of actions remain just that — gone are the additional triggers that manipulate what group of actions get a chance to select. A scan of the ultimate behavior system file deployed in *alphaWolf* shows that, in the absence of short-circuiting triggers that act in this fashion, the complexity of the action selection process, as measured by the number of triggers connected to all action tuples reduces by 60%. Additionally, it appears that several action tuples, that act as silent placeholders for actions that subsume control over both groups, would disappear entirely.

Secondly, and more speculatively, are the other hypothetical simplifications that Diagram could have offered the designers of *alphaWolf*. Most significant is the ability for the channel mechanism to represent, schedule and revoke sequences of actions. As I have stated, *page 82*, the creation of chains of actions and the deferral of actions while others run seemed fundamentally difficult in *alphaWolf* — difficult to both express, and to express without damaging the action-groups ability to prevent dithering. Sequences where pups explore their environment, by moving between random locations; where adults move away from the pack only to return later; and indeed the primary interaction of moving towards a pup, fighting it and winning or loosing form the backbone of this kind of direct-able agent's "scriptability". The deferral of directed escape while being attacked also suffers from being only indirectly expressible inside the wolf-pup's action system — a set of triggers that "latch" user interaction can be seen throughout the code. Finally, it might have been possible to unify the top-most layers of the wolf-pups motor system with parts of the action system in a single set of Diagram channels. The patterning of the motor programs of *alphaWolf* often reached points of extreme complexity due primarily to the navigational complexity of the environment (the need to move pups towards moving targets). If both scheduled actions and the resulting motor programs could have "seen" each other, and more importantly been seen by the navigation system, issues of persistence — either running actions until the pup actually arrived at a point where interaction could occur or eliminating attention switches made impossible by the current motor programs — could have been avoided.

While it remains, of course, impossible to verify these claims of the utility of the Diagram system for the *alphaWolf* installation it is equally clear that if these problems can arise even in a system in many respects ill-suited for their treatment, it will not be long before they arose in other works. Indeed, in *how long...* and *22* we shall see ghosts of the complexities of *alphaWolf* appear again — "modal" action systems that move through constrained phases, often patterned by the flow of time through the work; actions, be they the drawing of lines, or the manipulation of existing graphic elements, that unfold over small, scripted periods of time; and a general blurring of the boundary between action selection and motor patterning. Prior to deploying Diagram in ernest in my pieces for live dance, however, I completed an installation that was very much about the complex and "precise" patterning of time — a live composition, *Loops Score*.

Loops Score — live computational music for *Loops*

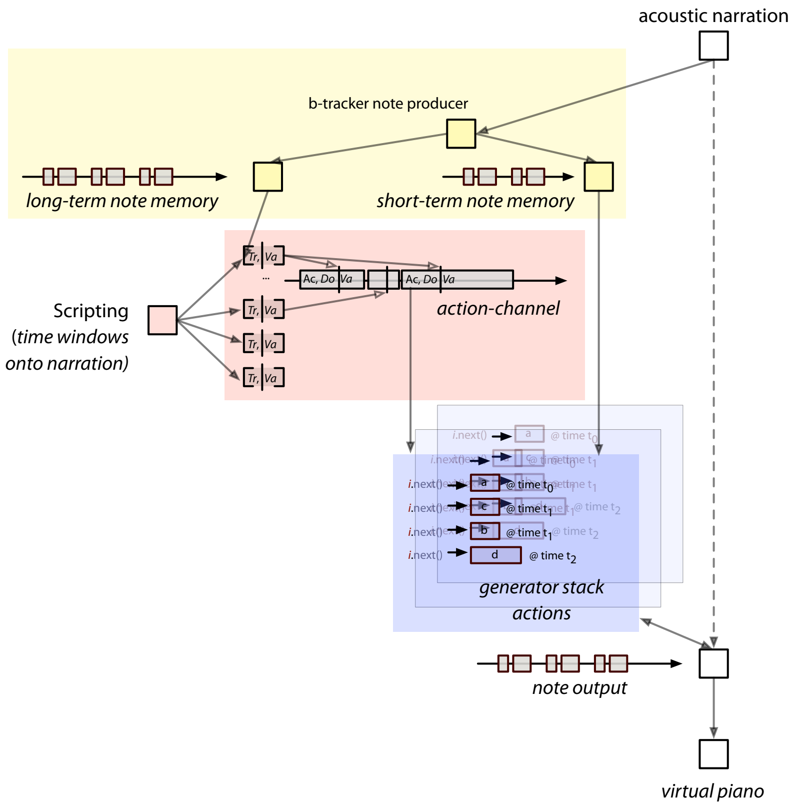


figure 73. *Loops Score* agent overview. Each of these boxes will be "unpacked" in the main body of the text.

Loops Score is the music that was made to accompany *Loops*. It was constructed by analogy with the visuals two years after *Loops* premiered. Just as *Loops* constructs a set of interacting processes that observe and recast the motion of Cunningham's hands, *Loops Score* takes a set of interacting musical processes that listen to and restate the sound and language of Cunningham's narration. Cunningham provided a twenty minute recording session, independent of the motion capture session, consisting of him reading from his 1937 diary — his first visit as a young man to New York. The sonic palette for the work was found in a high-fidelity sample set provided by the John Cage Foundation of a prepared piano, prepared in accordance with Cage's instructions accompanying his sonatas and interludes. By selection, filtration and pitch shifting, this prepared piano was turned into a set of seven pianos, each with a different, but radically expanded, timbral range.

Unlike *Loops* there is a single agent at work in the production, and, in contrast with *The Music Creatures* this agent has no visual form and its musical output comes without virtual movement analogies, but simply as a set of notes sent to a set of virtual pianos. However, *Loops Score* was a work that continued many of the technical themes that *Loops* started; in particular: what it might mean to score one or more action systems and how one might create a work that navigates alternating layered structures of emergence and control. In particular, the ability to general complex structures in channel / marker representations and then have competing tasks slice across these channels, organizing and culling them, is in itself the layer of open emergence and detailed specification.

The Music Creatures was a unique exploration of the possibilities of relationships between the body of an agent, the sound that it makes and the sound that it understands. But as far a mainstream computer music is concerned this appeal to virtual physicality was a digression. *Loops Score* approaches computer music more directly. Where *The Music Creatures* have agents with complex perceptual and motor skills and simple, almost script-like developmental action systems glued together with a few automatically learned parameters, *Loops Score* is on ground more common with the traditions of the interactive music work: the complexity is in the scoring, and in the interpretation of this score — specifically the "action system" of the agent. There is, no doubt, a future installation to be made that has comparable richness in each of the three parts of the agent decomposition, *page 421*, but taken together I believe that these works do a good job of creating a preliminary sketch of that installation's territory.

The open process score

Early in the creation of this piece we rejected the, perhaps rather Cunninghamesque, idea that the meaning of the words might be technically unrelated to the processes that act upon the sounds of them, opting instead to find the "process score" for the music out of the text itself. The search was to find structures in the text that had both musical and linguistic function, that were 'half way' between forms found in music and forms bearing linguistic meaning. A number of forms were found, and each form was constructed as a loose template. They included **lists** inside the text both large and small; **comparisons** and spatial relationships; markings of the **passage** of time; **returns** to previous locations.

These templates operate on the word level, and one could perhaps imagine given a robust enough speech-to-text system performing this template-matching in real time. This was not attempted and not required given the finite and predetermined narration, but remains an enticing possibility for future work. Rather, an analysis of the text of the narration was coupled to sound of the narration through marking of the onset and offset of each word in the narration. Thus for each present atom in each instance of each template, we can provide a time period this atom is "listening" to the sound of the narration.

Each of these structures, gathered from the text, marks a channel-based *script* which opens windows for actions to occur. Actions inside *Loops Score* are triggered by the presence of narration material inside these windows; in the end some 150 windows are marked on the score, cross-linked to sixty-five actions. The vocabulary of possible actions corresponds metaphorically to the kinds of structures that trigger them: **list-actions** repeat their triggering elements while searching for a stable rhythm inside the elements; **comparison-actions** state their elements and then emphasize the differences between them; **passage-actions** state their first element and then continue to look for material that is sonically related until the close of their marked passage; return-actions compress material from their triggering element all the way back to where they "came from". The actual programming of these musical cells is constructed using the manipulations of the marker / channel structure that will be described in the following section. These actions schedule the notes to be played into a channel which is itself exposed to the actions. Rather than "ballistically" writing the notes to be played to the scheduler, they opportunistically align, modify and perhaps even fight for space and relationships"on output".

Since the script is densely packed with overlapping processes, these actions compete using the action-selection mechanism described above. The budget for actions is dynamically set based on a smoothed version of the amount of musical material that eventually makes it out of the agent — providing a long-term, self-regulatory production.

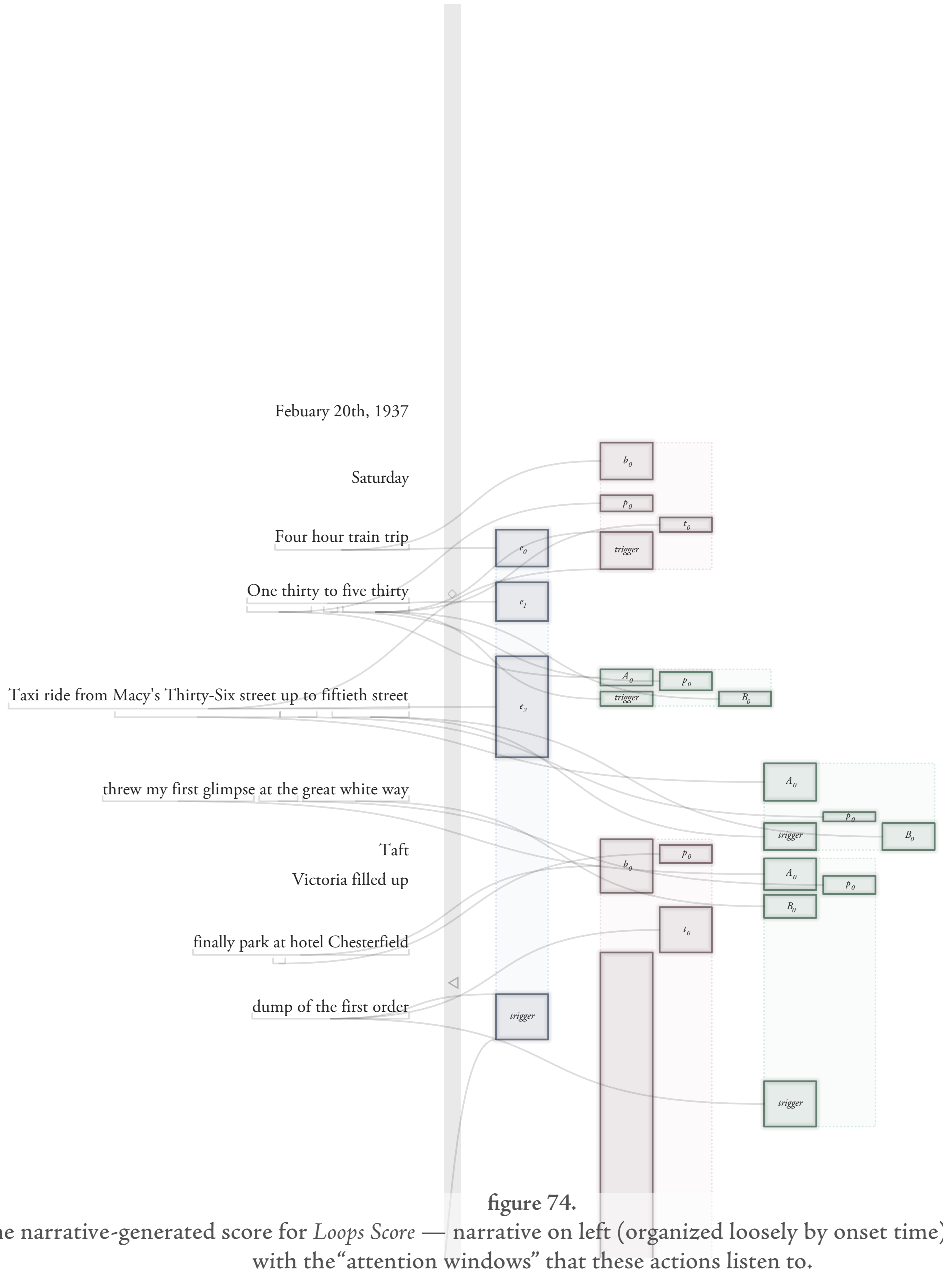

Febuary 20th, 1937
Saturday
Four hour train trip
One thirty to five thirty
Taxi ride from Macy's Thirty-Six street up to fiftieth street
threw my first glimpse at the great white way
Taft
Victoria filled up
finally park at hotel Chesterfield
dump of the first order
b0
p0
t0
trigger
e0
e1
e2
A0
B0

figure 74.
An extract from the narrative-generated score for *Loops Score* — narrative on left (organized loosely by onset time), potential actions on the right with the“attention windows” that these actions listen to.

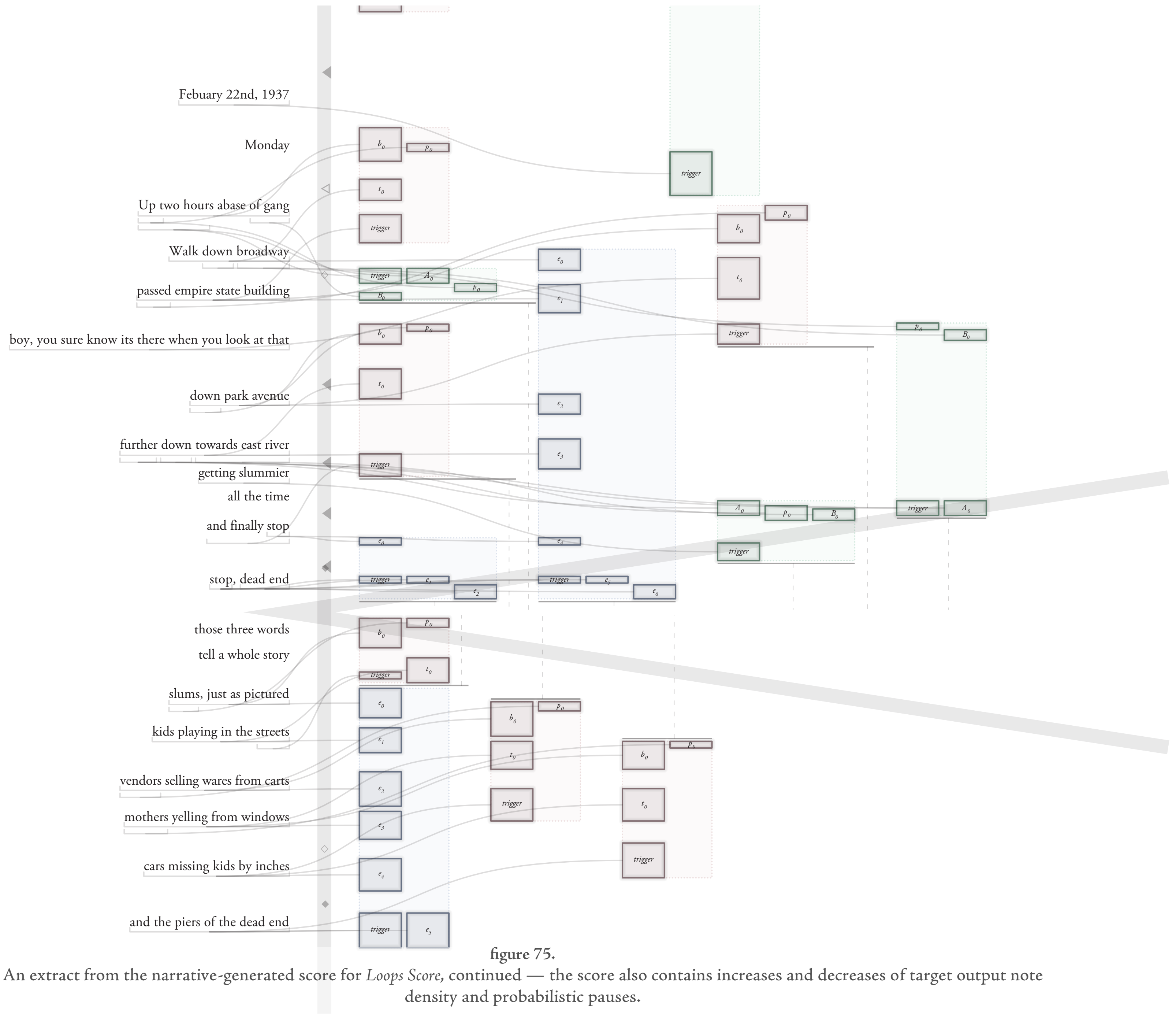

Febuary 22nd, 1937
Monday
Up two hours abase of gang
Walk down broadway
passed empire state building
boy, you sure know its there when you look at that
down park avenue
further down towards east river
getting slummier
all the time
and finally stop
stop, dead end
those three words
tell a whole story
slums, just as pictured
kids playing in the streets
vendors selling wares from carts
mothers yelling from windows
cars missing kids by inches
and the piers of the dead end
trigger
b_0
p_0
t_0
A_0
B_0
e_0
e_1
e_2
e_3
e_4
e_5
e_6

figure 75.
An extract from the narrative-generated score for *Loops Score*, continued — the score also contains increases and decreases of target output note density and probabilistic pauses.

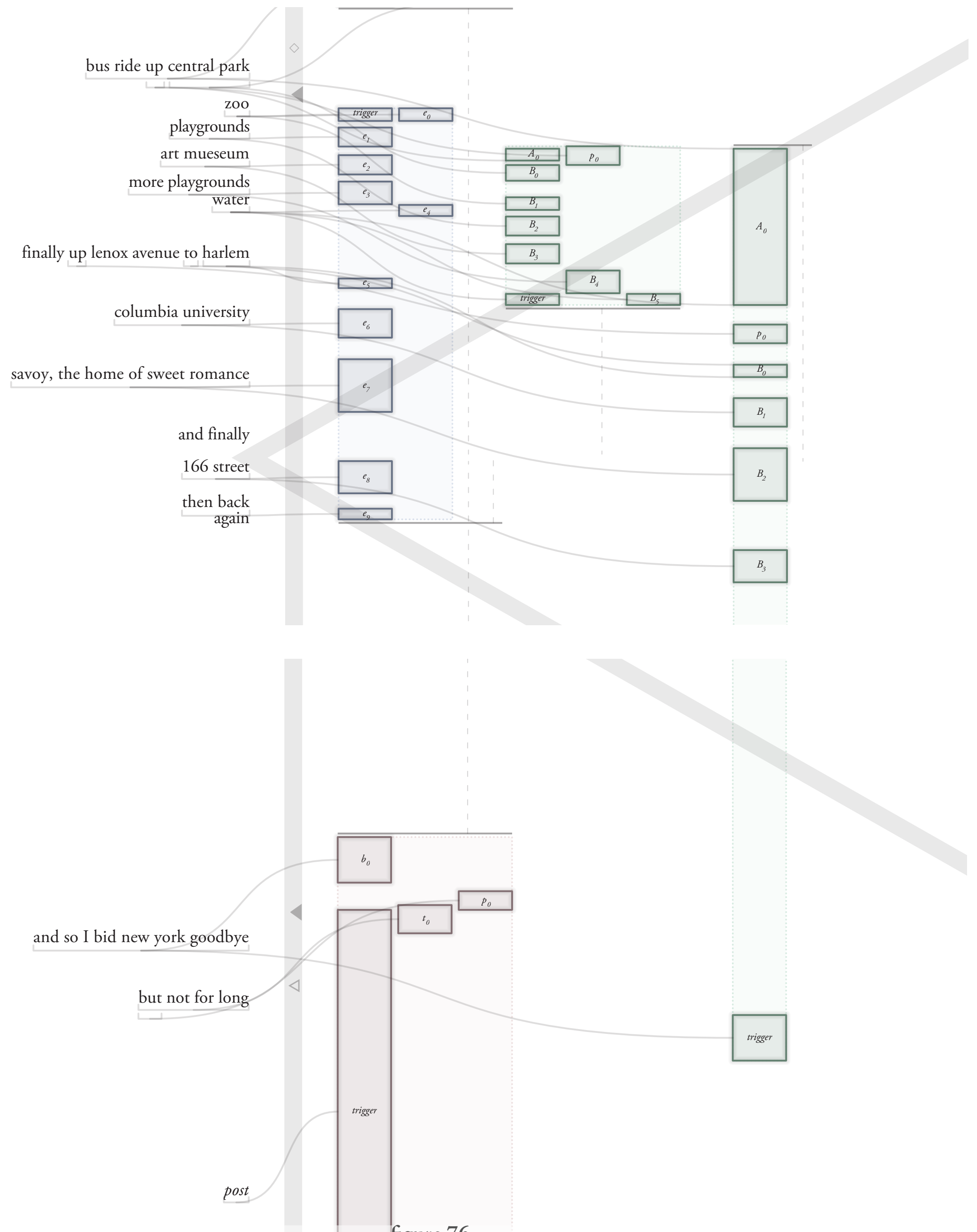


figure 76.
An extract from the narrative-generated score for *Loops Score*, continued — upon ending, the score loops. However, the memory of the notes played is made available to the processes in the next iteration, allowing actions to trigger in advance of all of their attention windows.

Thus, the agent of *Loops Score* exists in a perceptual world dominated by the sound of the narration annotated by this script. The perception system here transforms the audio into a series of overlapping note events. And we find ourselves again in a problem domain halfway between that of music and that of speech. The conversion of raw unstructured audio to quantized notes is a problem that has, of course, received considerable attention — it is in essence the inverse problem to the forward task of synthesizing sound from a score. And, particularly in monophonic worlds, solving this problem is often an important initial step in interactive music systems. However we are in an adjacent but different domain here — converting speech rather than music to musical notes — a less grounded domain. The coarsest version of this problem might be the extraction of prosodic contour and the segmentation of voiced and unvoiced parts of speech and this too has received some attention. Our goal then is more musical detail than that afforded by speech-based approaches, and less musical fidelity to a ground-truth with a more complex input than polyphonic music-based approaches.

For what I believe to be the state of the art in this problem, **A. T. Cemgil**, *Bayesian Music Transcription*, PhD dissertation, Radboud University of Nijmegen, 2004.

One of the earliest score followers used unstructured audio: **B. Vercoe, M. Puckette.** *Synthetic Rehearsal: Training the Synthetic Performer*. Proceedings of the 1985 International Computer Music Conference. San Francisco: Computer Music Association, 1985.

Because of our need for musical accuracy, we forsake the simplicities of a pure-monophonic pitch tracking-solution. We recast this perception problem as a tracking problem, tracking peaks on successive overlapping Fourier transform frames, seeding our b-tracker ongoing model population with the results of a lightweight monophonic pitch follower. We use the b-tracker perceptual framework to track these peaks and convert them into musical "notes".

This pitch tracker is an implementation of the lightweight voice-pitch tracker: **L. K. Saul, D. D. Lee, C. L. Isbell, Y. LeCun**, *Real time voice processing with audiovisual feedback : toward autonomous agents with perfect pitch*, Advances in Neural Information Processing Systems 15. NIPS 2002.

To use the b-tracker framework we need to supply the following underlying process details: individual tracking hypotheses are represented as individual frequencies F with frequency "velocities" and amplitudes; they predict that the next Fourier frame will contain a strong peak at the same frequency, and a fre-

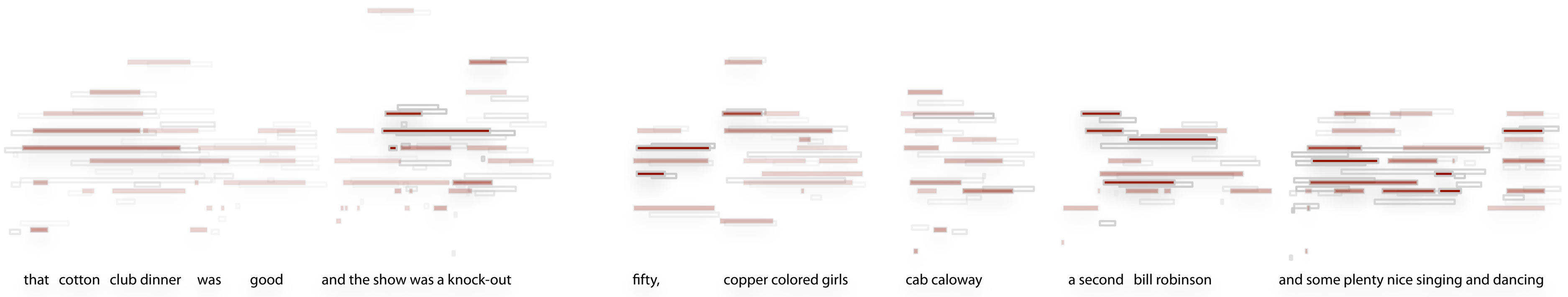


figure 77. The notes created (grey) and filtered (red) from the b-tracker analysis of the short-time Fourier transform windows from Cunninghams's narration.

quency one Fourier bin higher $F + R + Fv$ and one bin lower $F - R + Fv$; thus the untrimmed branching factor of the forward search is three. Successive Fourier frames are overlapped by a factor of four, to provide precise frequency information in the case that only one frequency is present in a bin. In the case that multiple frequencies move in and out of a bin we'd expect the kind of crossing hypotheses that can be disambiguated with the frequency velocity information. The monophonic pitch tracker constantly seeds the tracker with hypotheses at its output pitch, should it determine that voiced speech is actually taking place.

Once a hypothesis has survived the culling process of the b-tracker for four successive frames (four frames overlap a single location), it has the opportunity to emit a note. To do this we need to convert the pitch history of the hypothesis to a musical note. Since we have no underlying pitch grid from the underlying sonic material, we have to adapt one as we move along. We can represent this pitch grid as a set of hypotheses that get adapted by the pitches emitted by the lower level tracker.

Setting the size of the bins to be multiplicative increments of $2^{1/12}$ gives us a adaptive chromatic scale. Choosing larger increments gives us access to potentially interesting hybrid modes — *modes* because the scale is typically quantized

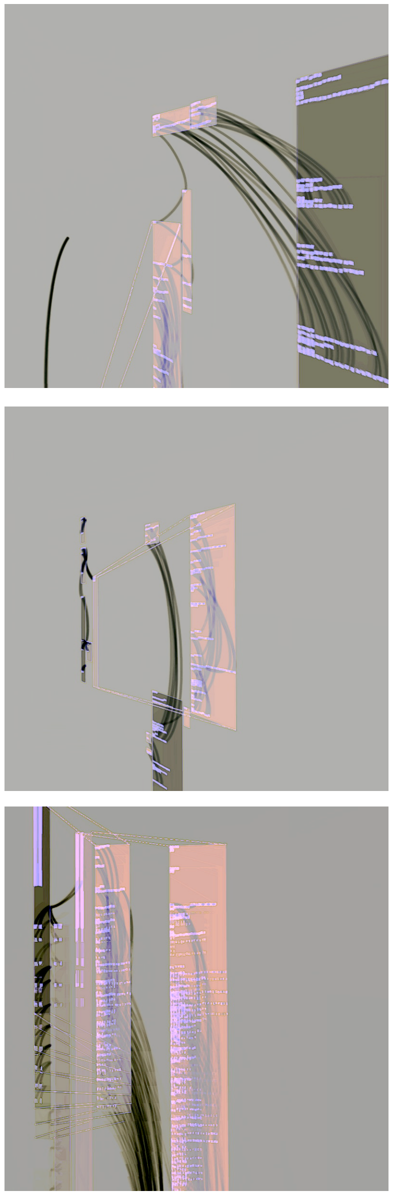


figure 78. The diagram visualizer (which showed alongside *Loops* and *Loops Score* in a separate, synchronized interactive kiosk). These images are quite literally the markers and channels generating the music.

more coarsely than a full evenly-tempered chromatic scale, *hybrid* because the locally coarse quantization grids are allowed to be globally misaligned. This increment, and the speed of adaptation / forgetting in this layer are free parameters. And in a few places the score forces a flushing of this memory — a modal break accompanies the change of day in the narration. This granularity is easily expressed as a "cleanup" process on the ongoing model stage of this b-tracker implementation, *page 173*.

These hypotheses, labeled with note values, amplitudes, and ultimately with durations, are the "output" of the perceptual layer of the *Loops Score* agent. These hypotheses are injected into a short-term memory (of around five seconds), which maintains the full merge histories of the hypotheses and a long term-memory (of around thirty minutes, twice the duration of the underling narration) which maintains just enough information to write a musical score.

This resulting post-perceptual surface is thus easily presented in the channel / marker representation. This is the raw musical material that triggers and enters the actions of *Loops Score* and my discussion will now turn to how the raw *algorithmic* material of these actions are constructed. These materials, although introduced here in a "computer music" use are all specific only to the marker / channel representation and the fundamental support that Diagram offers — they are *of* music, but not musical themselves. They form the basis for the odd "musicality" of movement and interaction that I have sought in my dance pieces and beyond.

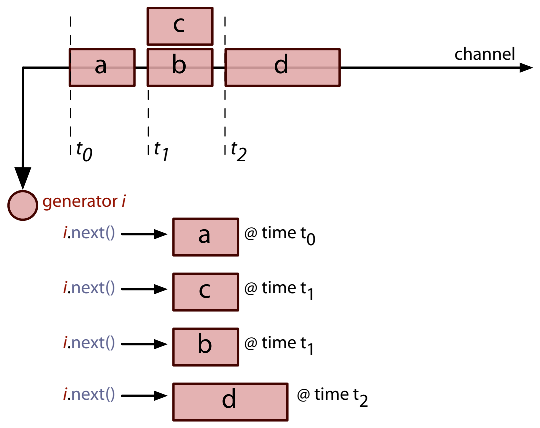


figure 79. Converting between channels and generators is easy. Here a generator "reads out" a channel in order.

The most important is a diagram channel *generator* — this is an extension of the co-routine / resource framework introduced for *The Music Creatures, page 140*. In that work the framework was constructed to allow small imperative programs to run concurrently while allowing cross-program interaction and dependency in the shape of resources. Typically, these small programs got one chance to execute (or "update(...)") per update cycle of the music creature in which they were embedded. Here we allow our co-routines to "return" a sequential series of diagram channel markers, and move programs forward not once per update cycle but until the markers that they start returning reach a certain point in the future. These programs are responsible for keeping the diagram channels' description of the future filled up and are called upon to generate more material, if they can, when they are needed.

Musical material generating processes, both simple and complex, can be written inside this style. For example (this time in Python; the Java is similar to the co-routines previously discussed):

```
def scale(start, step):
    n=start
    while true:
        n = n + step
        yield noteMarkerFor(n, quarterNote(n))
```

produces an ascending chromatic scale. Parallel and series composition of processes are achieved using the same kinds of continuation-composition that the co-routine / resource framework already allow.

```
def contrary_motion():
    yield parallel( (scale(0,1), scale(100,-1) )
```

Clearly, we can turn channels into generators quite simply (this generator is nothing more than sorted channel marker *iterator*), and we can turn generators into (potentially infinite) channels through rendering out the markers that appear from the generator until a sufficient time in the future is reached. Nothing prevents generators from returning markers that do not increase in time along the channel, although such a practice is discouraged. "Flattening" generators can be applied to processes that cannot produce their output in time order, with increasing levels of latency (i.e decreasing levels of minimum future) being introduced for increasing levels of safety.

Generators and channels are complementary: generator-level composition is particularly good at producing memory and time-efficient processing of action-level musical manipulation; channel-level manipulations are good at producing transformations that cut across many channels or many time epochs and are suitable for memories and intermediate buffers.

The goal was to create from such a language framework a system that allowed the rapid development of incremental, real-time musical processes that contain as little latency as is needed to maintain their computational integrity inside a dynamic environment but no less — a system that blends the interactivity of reactive systems and the complex multi-temporal scheduling and planning of material that more deliberative systems achieve.

To flesh out the description of how this framework was used for *Loops* Score, we need to give some kind of overview of both generator-and channel-level manipulators from which the actions of the agent were created. Since, as premiered, *Loops Score* used thirty-five primitive generators, eight channel manipulators together to make three versions (each a generator) of each of the four sub-linguistic templates described above, we should group these generators and manipulations together into some kind of taxonomy.

An **abstract balance** is a generator-level Diagram facility that takes a channel that is having markers added to it and filters these events such that a target event rate is met — it requires that something, perhaps the markers themselves, can provide a scalar *value* for a marker. We have seen something similar, but less dynamic, in the persistent long-term learning of *The Music Creatures, page 127.* The task is to find some horizontal partitioning cut-off for a channel such that the rate is maintained if markers that are "bigger" that this cut-off. If we have a sorted list of N markers m_n in the source channel we place the cut-off $c_{\alpha,N}$ that lets the top α fraction through at position αN or, more accurately:

$$c_{\alpha,N} = m_{\lfloor \alpha N \rfloor}(1 - \alpha N + \lfloor \alpha N \rfloor) + m_{\lceil \alpha N \rceil}(\alpha N - \lfloor \alpha N \rfloor)$$

In practice the distribution of marker-values isn't necessarily stationary. Handling arbitrary non-stationarities is, of course, arbitrarily hard. The following gives a variable forward momentum generated by recent history:

$$c'_{\alpha,N,\beta} = c_{\alpha,N} + (c_{\alpha,N/2} - c_{\alpha,N})\beta$$

for $\beta > 1$ this extrapolates out how the cut-off is changing with time. It's easy to modify this approach further to prefer more pessimistic or optimistic (or rather high-value or low-value) extrapolations.

The balances can turn a channel of valued-markers, which might represent actions or perceptual events, into event streams with particular general *rates*. Often, it seems, it is easy to come up with a good metric for perceptual events, but much harder to understand how this metric transforms the underlying temporal behavior of what it applied to. Without these indirect connections, such as the abstract-balance, one begins to start tuning the metric itself in response to

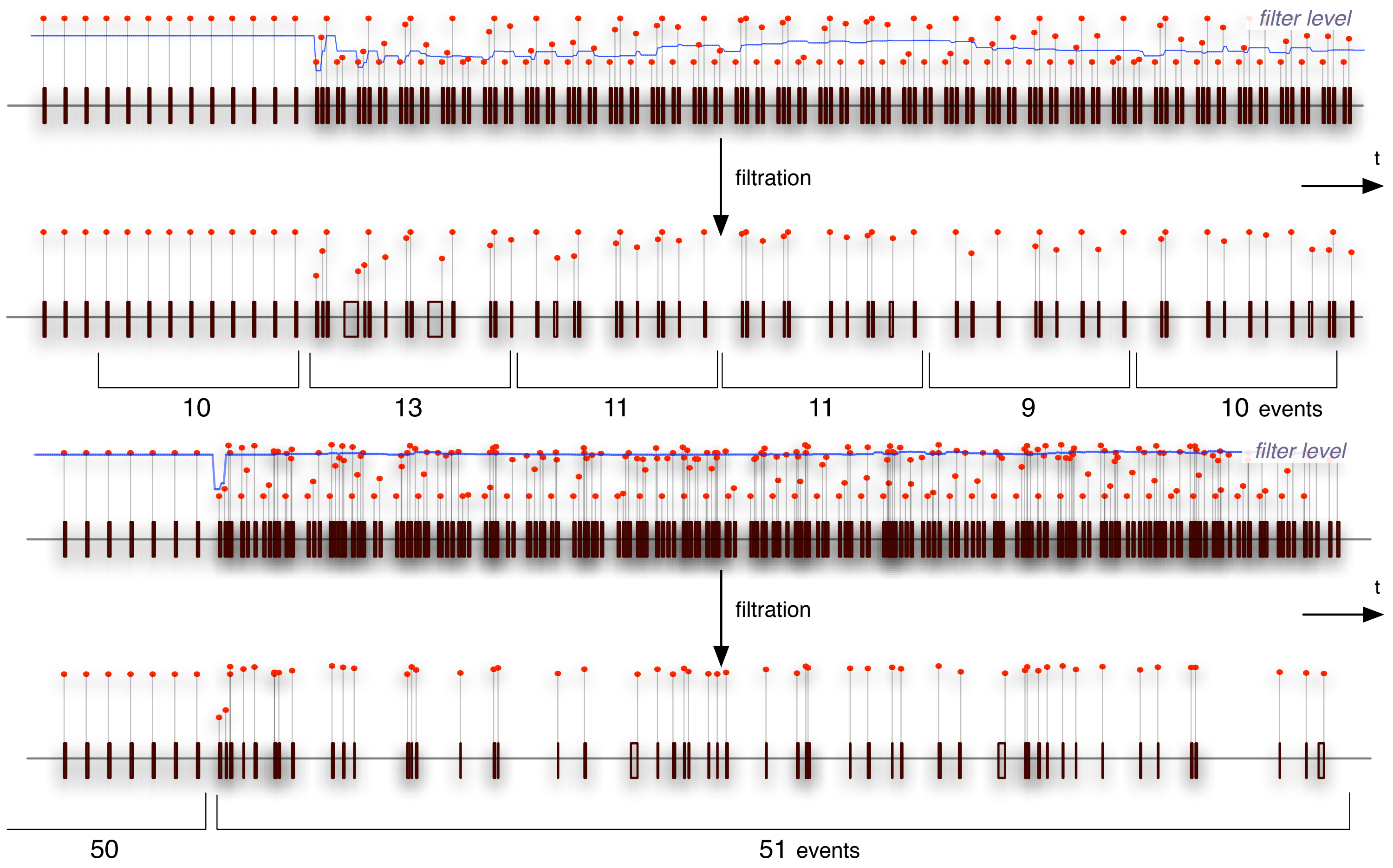


figure 80. The abstract balance correctly adapts its threshold to maintain a very event similar rate while capturing only the highest value events.

the temporal dynamics of the material that it is exposed to. That these connections should be more indirectly, yet more explicitly specified is argued strongly in the development of *The Music Creatures, page 159*. Such a blurring of principle (the metric) and pragmatic deployment (what it happens to be used on) is a generally unacceptable level of coupling between two parts of a system and is often how a commitment to a particular input or processing of input gets buried deep within a system that consumes it.

The complete abstract balance has two more parameters — a maximum latency and a minimum refractory period. The first controls the "look-ahead" window size and controls how far back in time the generator will be returning events. High latency will allow a better tracking of the target rate on highly non-stationary input. A minimum refractory period blocks events from being generated too close together, and, further, masks these events in the memory (the sorted array above) such that they do not feature in the computation of the threshold.

Generator-level operations - filtration / perceptual partial re-tracking

Another generator-level manipulation of the musical material captured from the transformation of Cunningham's narration is the perceptual stream tracker. This layers another level of the b-tracker framework on top of the musical material — a much coarser level than the original partial analysis was conducted on — that produces a number of generator streams of material. Each b-tracker hypothesis is a perceptual stream — a pitch value, and a pitch momentum. Although a number of attempts at musical perceptual stream segmentation are present in the literature, the existence of the b-tracker framework makes the implementation of another perceptual stream follower almost as simple as filling out a form: a hypothesis representation (pitch, pitch momentum) = (p,m), a distance metric between hypothesis and pitch class datum d $|p+m-d|$, a hypothesis predictor $p \leftarrow p+m, m \leftarrow \alpha m$. Now further generator-level manipulations can be performed on the lower (the lowest good hypothesis) and upper (the highest good hypothesis) lines of the transformed material.

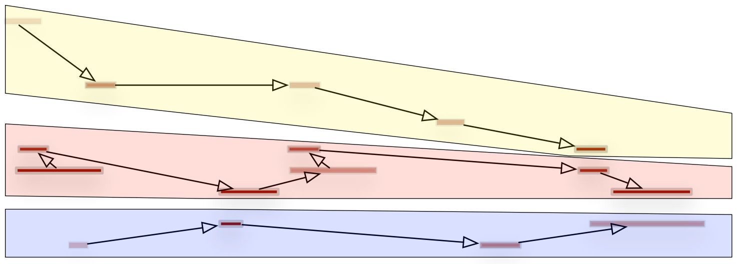


figure 81. A fragment from the short term memory of the *Loops Score* agent "re-tracked" by a new b-tracker process, segmented into three registers, corresponding to three b-tracker hypotheses. Note that no constant segmentation threshold would give this result.

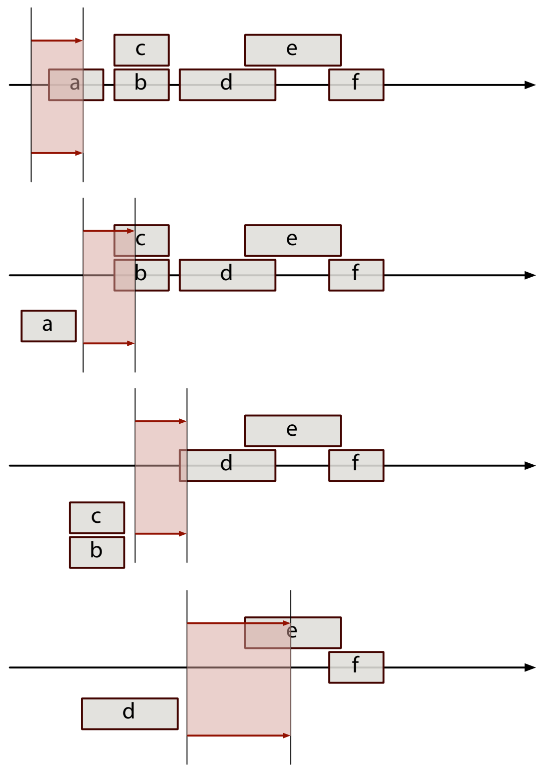


figure 82. The rolling culler is responsible for the past "forgetting" of markers in channels. It correctly handles both variable update rates and moving markers.

One simple, but ubiquitous channel-level manipulation is the rolling culler. This manipulation can be put to two uses, the first is to incrementally remove from a channel markers that have fallen past a particular time horizon, freeing up the space allocated to them and allowing the channel's memory to forget them. The second use for this rolling structure is to read out, according to some time-base the contents of the channel — like reading a score, (or playing a Fluid score, *page 373*) — by intersecting the time-base interval between updates with the contents of the channel. This is, therefore, one way of turning a fully laid-out channel into a marker generator. Rolling cullers are used to run the Diagram framework into a scheduler of music (with note events as markers) or an organizer of movement (with pose-graph or other instructions as markers).

In *Loops* Score, Diagram markers represent actions, planned out in time, that when traversed by the moment, result in notes being played, or parameters being set. The abstract balances are ways of culling sets of actions, based on criteria — metrics of value — thinning them out to a particular rate. There are other ways of thinning actions, that are more important if these channels are going to be coupled with other channels.

As a representation of future, scheduled actions, the Diagram marker channel appears to notate a case where actions are independent and that, barring any constraints, markers can be swapped or dropped independently. What if actions remained atomic, but were able to form molecules inside their channels?

A Diagram fusion filter takes a perspective onto the markers in a channel and looks for patterns that fit templates that, once matched, cause the replacement of the components with a new marker or markers. If we can find a succinct way of specifying these channel"chemistries" then they offer a powerful way of filtering or developing channel contents.

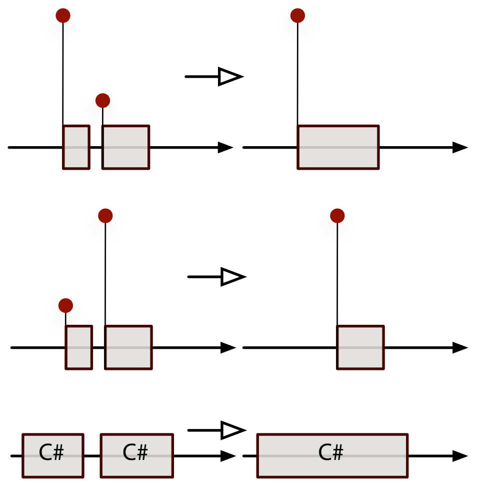


figure 83. Three example fusion filters take from the text.

We can write production templates for perceptual phenomena:

forward masking: $|_{t_0}\text{loud} + |_{t0+\Delta}\text{quiet} \rightarrow |_{t_0}\text{loud}$

backward masking: $|_{t_0}\text{quiet} + |_{t0+\Delta}\text{loud} \rightarrow |_{t0+\Delta}\text{loud}$

local quantization cleanup: $|_{t_0}\text{c\#}\, |_{t_0+\Delta}\text{c\#} \rightarrow |_{t_0+\Delta/2}\text{c\#}$

On the left-hand side there is a template to be matched — certain markers with certain properties with a particular temporal relationship (and, in particular, within a certain window of time); on the right hand side are the markers produced. This notation is not complete — it does not specify whether intervening markers are either ignored or prevent the match from occurring.

Programmatically we are free to construct these windowed template recognizers a number of ways and many recognizers are simply coded "by hand". We shall see later a less general, but more compact notation for describing these templates, *page 261*.

There are other abstract production calculi that are useful to define in general. One models a species of marker whose action is to set a variable to a value:

redundancy deletion: $[|_{t_0}(x \leftarrow a) + |_{t0+\Delta_0}(x \leftarrow b) + |_{t0+\Delta_1}(x \leftarrow a)] \rightarrow |_{t_0}(x \leftarrow a)$

for: $\forall a, b$

Such filters, applied over short windows, remove transient values that are set and then unset from a stream.

Such techniques fall very firmly within the domain of production systems and, as written, their use in either AI is far from new. However, in important previous uses of production rules in AI. the goal has been to create a complete system using this structure — essentially recasting the complete action selection and / or motor system problem in terms of competing or ordered production rule systems. In the framework here, the production system is not the material from which other system are constructed; rather they are available to other structures and representation. What is different here is the strategy not the tactics — the framework that this idea is embedded in and the principles that govern its deployment.

For example, the **C.L. Forgy**, *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*, Artificial Intelligence, 19, 1982 continues to have new implementers today: http://drools.org/Rete

The use of production systems *per se* in generative computer music is a little more ephemeral.

reversible, live and aware of time— no production rule deletes any material, even if it appears to "consume" its triggering markers, these markers remain in the channel (annotated as "consumed") and linked to the material that they produced and the production rule that coordinated it. This allows two important flexibilities: firstly, production rules that have fired are updated live when the details of their left hand sides are updated; secondly, The batching of notifications is one form of *deferred method execution*. Rather than, say, the addition of a marker to a channel causing the notification of every system interested in that channel we allow modifications to channels to be bracketed by calls to beginModification() and endModification(). Only at endModification() are notifications propagated (calls can be safely nested). One thing that is important to get correct in this implementation is how structural modifications — deletions and additions — should be handled. Many processes that execute on modifications are much easier to code if notifications are delayed until not only the addition of a marker to a channel has taken place, but the marker has finished having its attributes set and its relationship with other markers configured. Batching, as written above, achieves this delay.

However, for deletion of markers the issue is a little trickier. Again, consumers of notifications typically want to be informed of the impending deletion of a marker before the actual removal of the marker takes place, for it might have information stored with respect to the markers role in the channel. If these consumers only hear about the deletion after the marker's information has been deleted then they must be written to maintain a copy of all of that information. Maintaining this duplicate information, in turn, places a heavier burden on the notification mechanisms. So rather we make it easy to place the final removal of markers at endModification() time — there is a removeAtEndModification(*marker*). This defers final deletion until notifications have had a chance to act.

Finally, deletions need to be fused in the notification batch with any additions that take place to short-circuit notification cascades for markers that are added-changed-deleted or changed-deleted in one single batch. Ironically, such fusion of method calls would be an ideal use of the channel based fusion filters, were those filters not being constructed out of the use of this very notification mechanism.

production rules can choose to "un-fire" and delete their right-hand side should their conditions for acting cease to exist. This fluidity is vital in the case where the contents of the two sides of the production rule vary continuously, which is the case since the marker temporal positions are continuous, unlike the typical symbolic-level AI production systems.

a domain of low computational complexity — although in general even a carefully ordered set of production rules can become computationally burdensome (and a non-ordered set can, of course, run forever) we note that in most cases there are a number of factors on our side: the rules are not being used to structure particularly deep or broad computation — they are for cleaning, recognizing and embellishing small amounts of incrementally specified material, more complex computations can be achieved by other means; the computation is limited by the present time in one direction and by a configurable time horizon on the other — there is little point computing things far before the present, and far in the future can wait.

The computational difficulties that remain are ameliorated by careful batching of the notifications that cascade out from a channel when it is modified. Indeed, the batching of notifications becomes vital for keeping the whole Diagram system working at high speed.

high availability — one or more fusion filters can be attached to any channel; however, because these filters are often used to filter executable markers they are particularly useful for channels which represent destinations for deferred method calls. This is of such general applicability that we add to the context-tree annotation library for Java a new annotation tag (that triggers load time byte-code injection):

@deferIntoChannel(*channelDescription*, *parameterDescription*)

this tag marks a method (which must return void) as being deferred into a channel. Calling this method no longer results in the method body being executed; rather, a marker that will execute the method body is created and inserted into the channel at a particular time in the future. *channelDescription* points to a class that describes how to find the channel in question and the channel's time-base (this is typically either from the context tree or from a fields in this instance). *parameterDescription* points to a class that describes how to take the parameters to this method and transform them into attributes on the marker — attributes that will presumably be read and matched by fusion filters and other channel-level manipulations.

Fusion filters and @deferIntoChannel are the basis for an important part of my tool "Fluid". When visual elements (which can be though of as actions) are activated by multiple processes (which can be thought of as action-groups) we would like them to continue to see a stable life-cycle transition — start(), continue(), stop() and always in that order— despite multiple processes, spread throughout the code-base, starting them, continuing them and stopping them without coordination. The solution is to defer these start(), continue() and stop() calls into a channel and have a fusion filter clean up the overlapping messages into a diagram that expresses whether the action is in fact running or not.

$$|_{t_0}\text{start} + |_{t_0}\text{continue} \rightarrow |_{t_0}\text{continue}$$

$$|_{t_0}\text{stop} + |_{t_0}\text{continue} \rightarrow |_{t_0}\text{continue}$$

$$|_{t_0}\text{start} + |_{t_0}\text{stop} \rightarrow |_{t_0}\text{start}$$

If some latency in stopping and starting actions is acceptable we can elide stops and starts that occur too closely together into unbroken"continues":

$$|_{t_0}\text{stop} + |_{t_0+\Delta}\text{start} \rightarrow |_{t_0 \rightarrow t_0+\Delta}\text{continue}$$

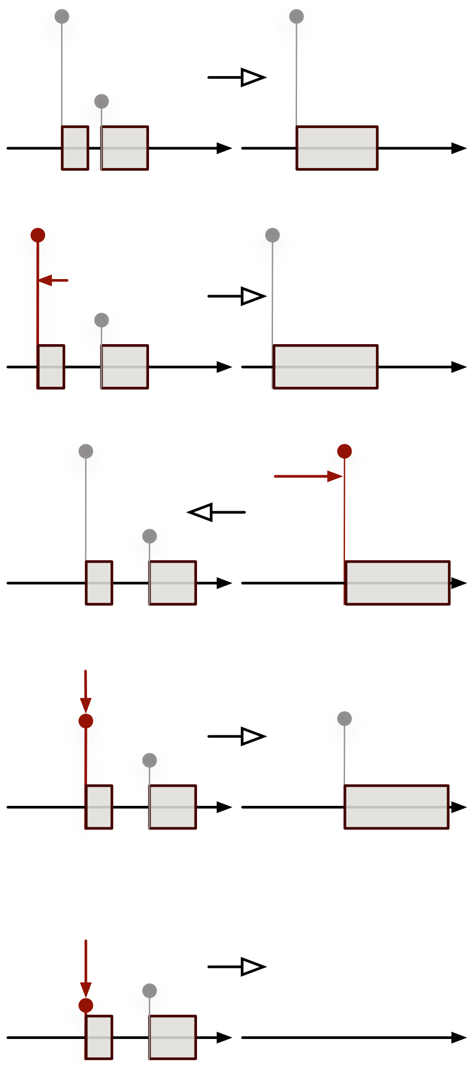


figure 84. By an intricate network of notifications, fusion filters are happy to collaborate with changing marker inputs, changing their output positions or attributes, or ultimately retracting their products altogether. The relationship is bidirectional.

This is useful, perhaps even vital, in the visual programming case where one needs to construct visual programs that are robust with respect to infinitesimal changes of visual element positioning, *page* 373. Away from the visual environment this offers a neat and self-contained (with respect to where the logic is located) solution to the problem of non-reentrant actions having multiple parent action groups. In *how long...* there is the *weaving* agent, *page* 351, whose"motor system" is entirely constructed out of such Diagram channels — the motor actions taken by the creature involve reorganizing a notation of the stage. The interface between the action system and the motor system evaporates, and only the programming language — the method call — remains.

Loops score uses fusion filters at a number of levels. Firstly, to further clean the output of the perceptual abstract balances, implementing coarse perceptual masking effects (for the processes that benefit from capturing a few strong and structurally important notes) and removing repeated overlapping notes from the transformation and sampling of the narrative. Secondly, to propagate — and, more importantly — repeat-with-modification, these samples of musical material.

Channel-level operations — modified time view

The narration for the *Loops Score* is not a linear story, but rather a series of independent fragments, between which there are pauses of random duration.

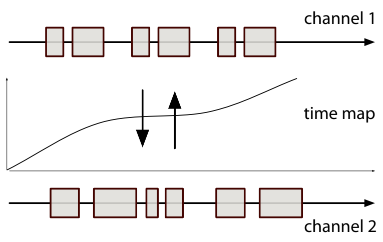


figure 85. Two channels coupled together reflect each other's contents, but present different temporal views — useful for the compressed memory structures of *Loops Score*.

The first channel-level manipulation is a simple one — the temporal distortion. This takes a channel and makes a view onto it that has a remapped time. With a bi-directional time remapping, both the view and the source channel are both live — markers can be added to either one and appear in the correct place in the other. *Loops Score* uses such a channel pair as its primary memory of its outputted musical events to compensate for these randomly generated pauses; the material that falls in the gaps between narratives is stretched or compressed when written and stretched or compressed again when recalled.

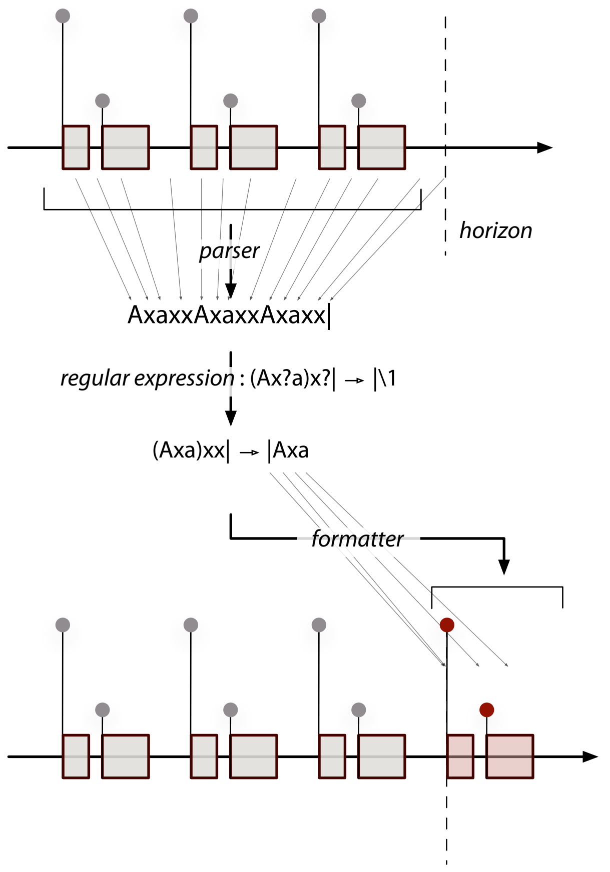


figure 86. An example "continuation momentum" written using a regular expression.

In the fusion filters above I loosely wrote "production equations" such as:

forward masking: $|_{t_0}\text{loud} + |_{t0+\Delta}\text{quiet} \rightarrow |_{t_0}\text{loud}$

We can code the recognizers (the left-hand side) and the producers (the right-hand side) by any means. The only constraints are that the recognizers must work incrementally — in response to batched notification updates from the channels — and producers should work non-destructively and reversibly — installing the request notification inside the source markers to maintain their relationship should their source markers change, or delete the production should their source markers be removed. However, for the sake of quick experimentation and tuning if nothing else, sometimes it is more conceptually useful to appeal to an intermediate and less general notation — particularly in the case of rhythmic cell generators.

So I will define a smaller set of channel listeners, which I will call channel momenta. These classes are responsible for taking the contents of a channel and continuing it out a little further in time. They are useful for maintaining a metrical grid, repeating otherwise idiomatic phrases for the purposes of proto-rhythmic structures. In *how long...* they are ways of injecting regularity into the movements of an agent (*accumulation, page* 329) without either choice or rigid duplication. In *Loops Score* they are responsible for continuing a motif such that other processes can effect the repeated gesture. Their left-hand side is always bounded by a temporal horizon (the "long past") of fixed size or number of elements and by the last element in the channel (the "now"). This window contains the pattern that is to be matched by the channel momentum object.

Of course, there is one pattern-matching domain where mainstream computer science can give us a significant head-start — so called "regular expression" matching. Thus there is an intermediate subspecies of fusion filter that acts on annotated strings using an extended regular expression library. If we can produce a channel *parser* that takes a windowed area of a channel and transduces the markers and the gaps between markers into strings of annotated characters, then we can formulate our production rules in terms of regular expressions. Most regular expressions (and all regular expression engines) match portions of strings of characters using expression encoded in other strings. In Diagram, the target characters are annotated with information connecting them to the markers, or the spaces that created them. These annotations do not affect the matching power of the regular expression, but they do follow their characters along for the ride. Produced markers can then be created in terms of the "captured groups" (strings of now annotated characters) of the template regular expressions by channel *formatters* which have an opposite role to channel parsers, turning these marker-annotated characters into new markers in the future of the channel.

In this work I was assisted greatly by the open source regular expression engine that ships with Java 1.5 — http://java.sun.com

Channel-level operations — opportunistic alignment

Most of the channel manipulations described above cut across multiple times on the same channel. There is an important class of channel manipulations that link markers across channels, typically markers that are temporally proximate. Of most use in *Loops Score* are the alignment manipulations. These take the markers in a source channel and try to make the markers in a target channel line up. At first blush, this is is not too dissimilar to the object-based alignment of popular graphical tools (such as Adobe Illustrator or Apple Keynote).

Adobe Systems Inc — http://www.adobe.com/illustrator

Apple Computer, Inc — http:// www.apple.com/keynote

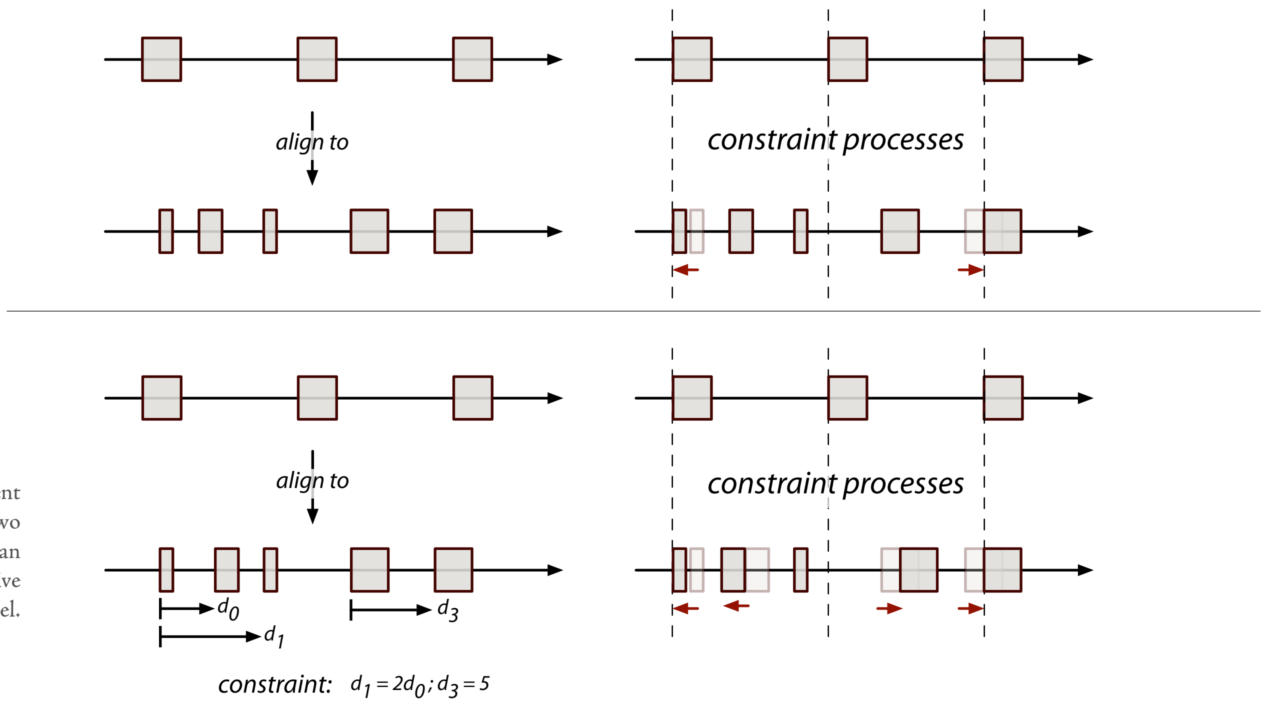


figure 87. Opportunistic alignment processes try to keep the contents of two channels aligned, within limits. They can work with any constrain processes active in the channel.

The fastest implementation is the most obvious: specifically, when a pair of source/target markers fall close together in time, we change the location of the target marker to align with the destination marker.

However, again, we return to the principles of the Diagram system. In particular, these alignments should be reversible and live. The key to maintaining these principles is to store the the marker positions in a framework that blends multiple, overlapping and persistent opinions about what a position should be. We have already seen one such framework — the generic radial-basis channel. Specifically, we create a class of marker that stores its position and duration in a generic radial-basis channel, *page 136*, that has a (position, duration) value representation. Now we can re-implement the above example. Rather than setting the location of the target marker to be the position of the destination marker we create a posting that expresses and maintains the constraint that the target marker should be the same as the source marker, whatever that value should be. Now external processes are free to move the source marker around and our pinned target marker will follow — therefore, we should also express the conditions in which the constraint is violated and "removed" (contributing to the radial-basis channel with zero weight). This makes the alignment reversible (no information is deleted) and live (the operation is actively maintained).

Such alignments are extensively used throughout *Loops Score* to manage and reduce the otherwise chaotic rhythmic complexity that otherwise comes from a number of independent musical processes taking musical material and subjecting them to repeated transformation. Indeed the alignment of markers before transmission to the virtual piano is the central source of rhythmic arbitration at the output "stage" of the agent. Unlike standard rhythmic quantization there is no global grid imposed upon the output, but a local complexity limit for the temporal patterning.

Four extensions make this alignment more useful and more detailed for musical purposes. Firstly, when aligning with extremely sparse channels one might wish to use a fusion filter to generate virtual markers "between" sparse markers. In our graphical tool metaphor this is equivalent to aligning not just to the edges of the page, but to the center as well. Secondly, we might choose to find the temporal distortion for un-aligned markers in terms of the nearby aligned markers. For the purposes of efficiency *Loops Score* uses a simple linear blend for un-aligned markers that fall between their nearest aligned markers; *Imagery for Jeux Deux* uses a a radial-basis function solution after a straight line fit to couple the note-level channel output of a score follower to a set of video keyframes. Thirdly, there is no reason not to make the constraint bi-directional; rather than forcing the target marker to have the same onset as the source, we force both markers to the same, intermediate time. This is the technique used in the output stage of *Loops Score* (where, additionally, the relative weights on the movement come from the amplitude of the musical event represented by the markers).

Finally, we note that in the generic radial-basis formulation we can also encode additional constraints into the positions and durations of the channel — perhaps some markers must be in the middle of others, often the ordering of markers are significant and cannot change. This representation will see a much more use when we discuss the visual counterpart to the channel marker: the Fluid graphical system.

Concluding remarks

Loops Score is a densely overlapped work of computer music that, like *Loops*, moves between areas of shocking clarity — piano mimicking the sound of Cunningham's voice — and periods that are propelled and sustained by its own obscured but palpable logic. Like *The Music Creatures*, the piece with its general strategy of capture and repetition-with-modification, produces clearly perceptible intentional development, a deeply rhythmic movement with no stable pulse or tonal center. More than *The Music Creatures*, it is an oddly self-balancing yet unbalanced music, culling the variety of the over-prepared prepared pianos and often offering slow and audible development of material.

Technically *Loops Score* set out to be an exploration of the recently created Diagram framework in a setting closer to traditional computer music than *The Music Creatures*. That it offered a thorough "work-out" of the technology at the lowest level is undeniable — the architecture survives the instantiation and destruction of tens of thousands of actions and perhaps millions of notes without intervention. The re-coupling of the strategies afforded by this work and a more visual, perhaps a more "agent-like", set of concerns occurs in the new dance pieces, in particular the work for *how long...*

I believe that in this virtual choreographic domain the emphasis, borrowed from my musical concerns, on the complex patterning *of* time and the authorship of rich, open forms *in* time provides a fresh perspective on the creation of live time-based media. The Diagram framework successfully hybridizing the reactive, or "interactionist", tendencies of shared by both the agent and mapping perspectives with the more ponderous and typically off-line strategies of non-real time and perhaps even non-algorithmic music.

We should step back and return briefly to the axial decomposition of action-selection techniques discussed earlier, *page 73*, so see where the diagram framework fits in. Clearly, the Diagram framework core action-selection algorithm pushes the c43/c5m action strategy into allowing multiple simultaneous actions. However, as a supporting framework, Diagram also reduces the burden of responsibility on the core selection technique — no longer is it responsible for all of the high level temporal patterning of an agents actions. No longer is action selection the final structuring step (with the details to be filled in by a motor system) in the creating of temporal structures. Rather it is the first step, with the results of action selection to be further crafted by collaborating processes. By allowing post-hoc manipulations of future, scheduled actions and interactions, multiple processes that cut across the results of action selection can both clean up and constrain them. This acts to reduce the "temporal uncertainty" faced by the author of an agent, while at the same time offering hybrid strategies, orthogonal to action selection, that allow for more complex patterning of action.

In standing back and surveying *Loops Score*, the Diagram framework and the smaller "glue systems" that this chapter has developed, we see that this "complex patterning of action" — what I might call the *choreography* of the digital agent's actions — is in fact the goal of this family of techniques.

The context-tree permits the kind of modularity that defuses one of the central methodological contradictions of making art with a complex process — how choices act upon a process that is simultaneously being developed, how the *names* of parameters, styles, behaviors, states, movements retain power over the complex processes without fusing solid the agent's inner workings (and with them our creative process) prematurely. Through decoupling elements, through code injection, and through carefully made persistence frameworks, we have complex processes that can support long-term collaborative practices. I am tempted to call this the complex patterning of *collaboration*.

Is this choreographic? The Diagram framework, in its explicit articulation of action selection and scheduling, makes a computational *representation*, in the sense of the introductory chapter — a site for further, agglomerative, transformation, for the collision of processes and constraints. It is these structures that allow for the small, nimble agents of my work in dance theater.

These "language interventions", the multiple uses of the context tree, the agents made up of changing parts, the explicit patterning of action in the diagram framework, all are motivated by the need to author and contain intricate processes. One cloud remains, however — how to take the techniques developed in this chapter and cast them in such a way that they can truly be deployed "live", as it were, in a collaborative creative process. This thesis's answer to this will be given in the last chapter, on *Fluid*, the graphical environment that responds to this problem.