This chapter is necessarily more technical than the chapter that preceded, for it articulates the specific technical context and technical origin of my work. It introduces the agent toolkit developed by the Synthetic Characters Group in 2001, and includes an overview of the c43/c5 action-selection approach, as an important example of the agent as an "organizing framework". And it ends with a critique of the largest, most complex installation from that period — alphaWolf *— indicating how the contributions of this thesis respond to the weaknesses that appeared during the development of this work.*

Chapter 2 — Beginnings

In 1999-2003 the Synthetic Characters Group, of which I was a part, proposed a new line of action-selection algorithms and an action representation that is used in, or at least relevant to, many of the works described in this thesis. The agent toolkits that embedded these techniques were referred to as the 'c' series, c2 all the way through to c43 and c5 and these are as good a name as any to refer to the successive developments of the approach. This work perhaps reached its apogee with *Dobie*, an interactive synthetic dog character, trainable in many of the ways that real dogs are. Inside *Dobie* there is an elaborated version of the action selection structure used in the projects *alphaWolf*, *Loops* and an early version of *The Music Creatures*, and this is the action-selection approach that motivates the later discussion of the Diagram system.

The 'c' series of work is described in the following two papers:

R. Burke, D. Isla, M. Downie, Y. Ivanov, and **B. Blumberg,** *CreatureSmarts: The Art and Architecture of a Virtual Brain.* Proceedings of the Game Developers Conference: 147-166. 2001.

D. Isla, R. Burke, M. Downie, and **B. Blumberg.** *A Layered Brain Architecture for Synthetic Creatures.* Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI). 2001.

Issues of learning and adaptation inside this series are discussed in:

B. Blumberg, M. Downie, Y. Ivanov, M. Berlin, M.P. Johnson, B. Tomlinson, *Integrated learning for interactive synthetic characters.* SIGGRAPH 2002. Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM, 2002.

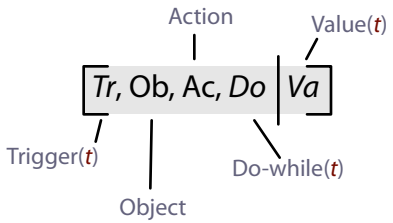

figure 1. The action-tuple consists of up to five parts. The **trigger**, **do-while** and **value** can each appear as a time varying scalar value.

The c5 action system begins with two structures, the action-tuple and the action group. The action-tuple, which is either active or inactive at any given moment of time, is a unit of action that aggregates the following components: each tuple has at least a **trigger** — an instantaneous scalar value that indicates how relevant or important the activation of action-tuple is to the current situation, in the broadest sense; a **do-while** — an instantaneous scalar value that indicates whether or not this action, should it be active now, is still important (high value) or is "finished" in some way (low value), and thus should become inactive; an **action "payload"** — the code that, while the action-tuple is active, should be executed; a **value** — a scalar representation of the value that this action has to the creature, this may be set by hand or learnt. Additionally, many action-tuples possess an **object** — code that can scan the perceptual system to provide an object for the action-payload to operate upon. An action-tuple can be read as a (fuzzy) rule, in the case of *Dobie* for example: when the perception system hears the trainer say "sit" (trigger), tell the motor system to sit (action-payload) near the trainer (object), and keep telling the motor system to sit until it has done so (do-while).

In this way, triggers, do-whiles and objects connect the perception system to action-tuples; the action-tuples when applied send commands to the motor system. The communication for each of these connections is supported by the "working memory" blackboard introduced earlier.

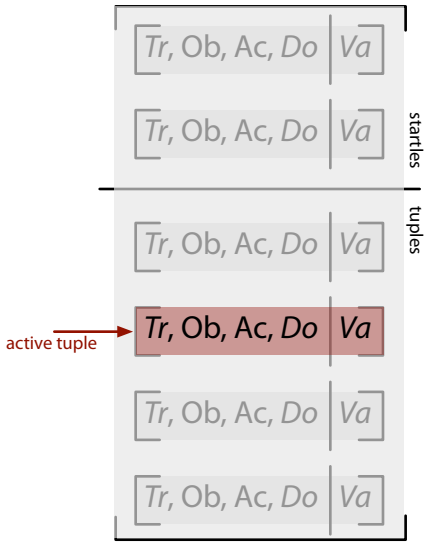

figure 2. Action-tuples compete for expression inside action groups. At any given time an action group has an active tuple.

Action-tuples compete for expression (activation) in action groups based on their *expected* values. An expected value is a combination (a multiplication) of the value of a tuple and its trigger, if it is not active; or its value and its do-while if it is. This captures a sense that a tuple should be both relevant (high trigger) and valuable (high value).

Each action-tuple is inside one and only one action group and each action group has only one and always one active action-tuple. An action group is responsible for reading the triggers and values of the actions, and the do-while of the active action, and deciding whether to keep the current active action or switch to another one. It is possible to implement an action group in a number of ways, but a good action-group implementation shares the responsibility with the action-tuples it maintains for the behavioral relevance, persistence and coherence of the creature — balancing the short term reactivity and opportunism of a creature with the medium term need to keep at an action active for long enough to see some payoff and the long term need to demonstrate goal directness and exploration of the behavior space. An initial action-selection algorithm, that forms the basis for most c43 / c5 characters, is as follows →

The c43 Action Selection Algorithm

state

startles — a list of action-tuples that get special privilege to interrupt others.

tuples — a list of action-tuples inside this group

currentlyActive — the currently active tuple

lastValues — a mapping from tuple to real number

algorithm

if the maximum *tuple*.expectedValueOf() over all of *startles* is greater than zero then the greatest becomes *nextActive*.

▸ otherwise,

construct the new map *nextValues*[*tuple*] = *tuple*.expectedValueOf() for all *tuples*

if any tuple that isn't *currentlyActive* has *nextValue*[*tuple*]>*lastValues*[*tuple*] and 2* *nextValue*[*tuple*] > *currentlyActive* .expectedValueOf() then select a new action

if *currentlyActive* .expectedValueOf()=0 then select a new action

▸ if we need to select a new action:

if all of *nextValues*[...] = 0 then nothing is done, and *nextActive* = *currentlyActive*

▸ otherwise,

sample *nextActive* from a normalized version of *nextValues*[...]

and set *lastValues*[...] = *nextValues*[...]

For example, compare the Scoot Framework: **S-Y. Yoon, B. Blumberg, G. Schneider,** *Motivation Driven Learning for Interactive Synthetic Characters*. Proceedings of 4th International Conference on Autonomous Agents. 2000.

or the work in **P. Maes,** *How To Do The Right Thing*, MIT AI Lab, Memo 1180, December 1998.

This action selection mechanism was born of three goals — each a desires for a certain simplicity. The first is to actually reduce the amount of action selection that takes place. Unlike other, more continuous strategies, the whole action system isn't exposed to a re-selection with every iteration; a new action is chosen only in two cases: that the current one has decided itself that it is done (and thus reduced it's do-while to zero or a low value), or another action has become much more important than it was the last time selection occurred. Indeed this choice moves to increase the ease with which temporal patterns can be constructed inside this action selection system. Therefore, this dramatic-thinning out of the temporal complexity of the action system is not for computational efficiency — it is for pragmatic clarity.

This is an authorship position. A behavior author can think through the behavior system they are creating for a longer period of time if it only changes at a small number of well defined situations. And, in principle, since the short term temporal dynamics of triggers and do-whiles matter much less in this framework than in others, the author is freed from the uncertainty associated with the degrees of "freedom" afforded by time-constants and gains. Some of the "emergent" behavior that would have been indirectly specified by these filter-constants are set directly, most notably by the structure of the do-while. The do-while clause enables individual action-tuples to finesse their long term likely-hood of preemption by other actions in the system, offer again a more local, and more explicit, control over the temporal dynamics of the group.

The second impetus for the c43 action-selection algorithm is a acknowledgment that some actions need to be handled differently: specifically, the short, transient, but always important actions are set aside into a separate group and have the opportunity to override the current action at any time. In *Dobie*, these *startles* control the initial introductory sequence and the reward marker; in *alphaWolf*, user interaction and reactions to pain; in an early iteration of *The Music*

Creatures, unexpected sound. This two-level approach decouples the trigger of these transient actions, which often form the fundamentals from which the interaction with the character is constructed, from the rest of the group, preventing an"arms-race" between the values and triggers of these two species of tuples.

The third motivation for c43 is a realization that the actions themselves, if given a stable, long duration life-cycle and a supple motor system can form quite large blocks and that there is little to be gained, other than a certain academic cachet, by making an action system out of completely homogeneous material.

There are a number of extensions to this selection framework that are important, and a number of ways that it directly or indirectly supports learning and adaptation of its parameters.

Hierarchical structure

With a little care, we can create an action-tuple that has, as its action payload, an action-group. Care is required because it is not initially obvious which one of the many ways of choosing how to generate the external interface — trigger, value — of an action-tuple from the external interfaces of all of the action-tuples inside a group one should pick, if any. Synthesizing some mean combination of triggers and values to present to the layer above endangers two advantages that a hierarchical action system might have — a more complete functional separation between hierarchical descendants that do not share an ancestor, and computational efficiency through partial evaluation of the hierarchical action tree.

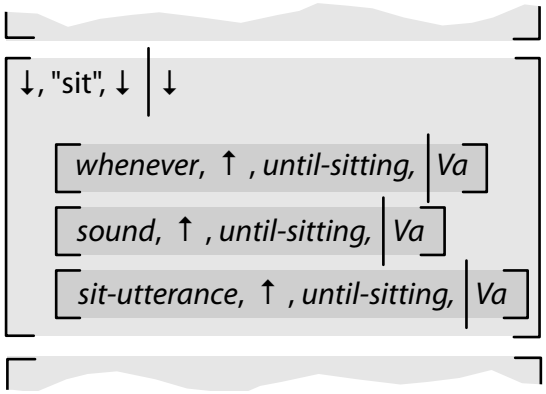

figure 3. Action-tuples can contain action groups. In *Dobie*, functional groupings are developed. Here is a group of actions inside a tuple that share the same action.

These advantages can be restored if triggers for whole groups can be specified ahead of time — that these triggers have something to do with the functional grouping expressed through the hierarchy is not too onerous a requirement — and that values for groups can be adapted on the same time-scales as their chil-

dren.

That said, there are characters constructed within the c5 thread that exploit this hierarchical flexibility in both of these ways. In *Dobie*, action-groups collate actions together according to action-payload and present triggers and values to parenting groups by computing maximum of its descendants. *Loops* chooses the second path and groups behaviors together for the purposes of scripting control, creating a semi-modal action-selection structure. *alphaWolf* runs multiple action -groups simultaneously, and contains a few actions that themselves do arbitration, using this second approach.

Complex value

The value part of the action-tuple offers one hook for adaptation to occur. In classic reinforcement learning we find a decomposition of the action selection problem that is not unlike the action-tuple representation. We can draw a table with rows representing actions and columns (perceptual) states. If we were to instantiate this complete table inside an action-tuple setting we would have one action-tuple for each cell in the table, and this action-tuple would have a trigger connected to the perceptual state and an action-payload corresponding to the action. Reinforcement algorithms exist that explore this table and learn the value of each of the cells.

The classic overview of reinforcement learning remains **R. S. Sutton, A. G. Barto,** *Reinforcement Learning,* MIT Press, 1998.

For example, in **q-learning,** for a transition taken by performing action $a \in \{a_i\}$ in state $s \in \{s_j\}$ that results in being in state s' we update the cell $\{a,s\}$ as follows:

$$q_{\{a,s\}} \leftarrow q_{\{a,s\}} + \beta\left(r + \gamma \cdot \max_j \left[q_{\{a_j,s'\}} - q_{\{a,s\}}\right]\right)$$

with β the learning rate and γ a "discount factor" and r the reward signal.

This cell is perhaps seeded with certain actions that are of high value (those that satisfy the motivations of the creature, for example receiving food). The intuition here is that action/state pairings that result in the opportunity to take high value actions are themselves valuable and during action/state transitions, during, that is, credit assignment these action/state pairings have some fraction of the expected reward propagated back to them.

Inside the action-tuple-based c43 architecture, this table is forever implicit, and we are free to partially instantiate it; the temporal dynamics of the action selection mechanism is significantly richer. More significantly, for the purposes of more biologically plausible and more authorable learning, we reject q-learning's proposal to continuously update a scalar value for each cell. Rather in *Dobie* we represent "value" with a slightly more complex structure (that, for the purposes of action selection may be turned into a scalar at any moment). This structure keeps track of an expected rate of reward — it remembers that a highly rewarding action is taken after a particular action-tuple at a certain *rate*. If the value of subsequent action is in keeping with this *expectation*, then no value update or propagation needs take place.

The specific details of these three forms of learning in the case of *Dobie* are well described in the paper summarizing this collaborative work:

B. Blumberg, M. Downie, Y. Ivanov, M. Berlin, M.P. Johnson, B. Tomlinson, *Integrated learning for interactive synthetic characters*. SIGGRAPH 2002. Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM, 2002.

The motivation for this simplification is in keeping with c43's rejection of individual update cycle dynamics — value propagation does not occur unless something changes in the world, no more than action selection occurs unless something (triggers or do-whiles) changes inside the action-group.

In *Loops* the values of action-tuples are still scalar, but the value interface becomes a window onto a looping script which modifies the values of the action-tuples that read from it (and provides a mechanism for the storage of active tuples), *page 98*. In *alphaWolf* the values of certain actions are coupled to a social-learning memory system developed by synthetic characters group-member Bill Tomlinson.

B. Tomlinson, *Synthetic Social Relationships for Computational Entities*. PhD Thesis, MIT, June 2002.

figure 4.
Dobie, 2002-3,
The Synthetic Characters Group.

Finally, we look at a powerful species of learning that characters can perform by dynamically extending its action system during the life of the creature in response to training. In the reinforcement-learning example above the rows (actions) and columns (perceptual states) of the "table" were set initially and the learning algorithm converged on a set of entries for the cells. This solution works quickly only if there are small number of rows and columns, otherwise the time to convergence quickly becomes prohibitively large. Using the sparse nature of the action-tuple representation, *Dobie* hierarchically explores both his state space (the number and organization of columns), his action space (the number and contents of the rows) and the set of pairings worth exploring (whether or not there even is a cell, or here an action-tuple).

In *Dobie*, action-tuples, which start with very general rules that have constant (and thus completely non-informative) triggers, maintain statistics about what finer-grained models in the perception system have better predictive power over obtaining reward. Eventually, new action-tuples, grouped automatically with their more general hypotheses, are created which represent these less general, but more reliable statements of the "rules of the world" for the creature. In this way, new state-action pairs are created through reliability-based hierarchical descent of the perception system structures. *Dobie* might join "sit" (action) "whenever" (trigger) with a more specific tuple of "sit" (action) "whenever there is sound".

Similarly, in response to reward signals inside the action system, the perception systems can begin to create finer models of parts of the perceptual world — children of "whenever there is sound" might be specific models of speech uttered by the trainer, labeled by the actions that potentially make use of these models. These new perceptual states become candidates for further state-action pairing

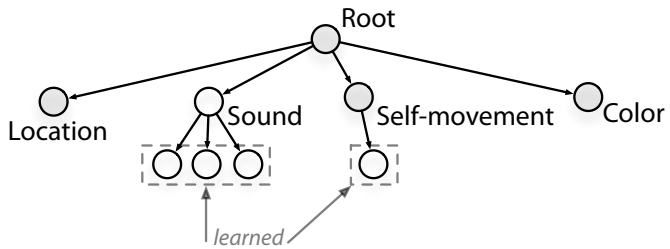

figure 5. A c43/c5's hierarchical decomposition of the perceptual world can be dynamically grown in response to reward signals or other forms of learning.

in the action-system. *Dobie* might end up with a tuple "sit" (action) "whenever the trainer says 'sit'", having started out knowing how to sit (and the unfading joys of food). Sometimes this modeling of the world is less driven by reward and more by the perceptions themselves: in *The Music Creatures* we shall see a few creatures that opportunistically construct other models of heard sound and — when they do so they store this "knowledge" (which is inherently procedural, it is the ability to *recognize again*) in the perception tree.

Finally, we can build the equivalent of perceptual models of the results of performing by actions that are parameterized in some way — for example if there is an action which corresponds to a complex selection and blending of motor-system material that is directed towards the goal of getting *Dobie*'s nose as close to a point in space as possible (perhaps some food). By keeping track of the specific animation patterns that *Dobie* ends up using to execute this action, if this action results in reward then we can generate new actions that correspond to the performance of these animations. *Dobie* might end up with a tuple " 'roll over' " (action) "whenever" (trigger), having started out with the motor competence, but not the action-system representation for rolling over. Similar learning, in a simpler domain, takes place in *Loops*: the learning of blended rendering parameters. In this case the new perceptual models are saved for later as part of the authorship story — during the "performance" of *Loops* as a work, no learning takes place, *page 107*. *The Music Creatures* store new perceptual models of action directly inside the motor system themselves, *page 145*.

2. The generic pose-graph motor system

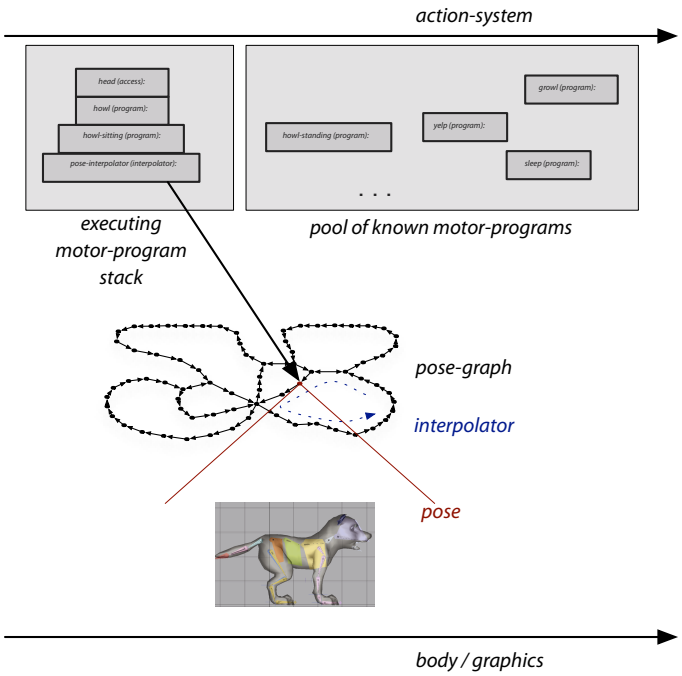

figure 6. The pose-graph provides much structure for the motor systems of agents. The action system communicates with a pool of "motor programs", in the most basic case this produces a stack of executable programs. These programs in turn call upon interpolators that play back paths through the pose-graph structure.

If the perception system of an agent is where the demands, time-scales and structures of the world first meet the needs, flows and internal preferences of the agent, the agent's motor system occupies a similar, but reversed role. Interpreting the selections made by an action system into a control structure, perhaps a control *plan*, executing it, and monitoring and reporting on its progress.

An action system may ask a virtual dog to walk over to an object in the world and "sit", it's a motor system that will be responsible for the *sequencing* of animation material onto the body representation of the creature in order to cause movement and ultimately sitting. Lower levels of the motor system ought to be able to answer questions concerning whether another step or walk cycle should be taken — knowledge about the *constraints* of the control of the body are stored here. And, as we shall see, the motor system as a whole ought to be able to answer questions concerning the outward appearance of the character — does our dog appear to be sitting or is it "standing up" — knowledge about the *contents* of the body representation are also exposed at this location.

My master's thesis: **M. Downie,** *behavior, animation, music: the music and movement of synthetic characters.* S. M. Thesis, January, 2000.

Graph structures in computer graphics are by no means unique in motor-system issues. Prior to my thesis:

C. Rose, *Verbs and Adverbs: Multidimensional Motion Interpolation Using Radial-Basis Functions.* Department of Computer Science. Princeton, NJ, Princeton University. 1999.

And more recently, **L. Kovar, M. Gleicher, F. Pighin,** *Motion Graphs.* 2002. Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM, 2002.

and on the automatic generation of graphs: **L. Kovar** and **M. Gleicher,** *Automated Extraction and Parameterization of Motions in Large Data Sets.* Transactions on Graphics, 23, 3. SIGGRAPH 2004.

The italicized words above — planning, sequencing, constraints and contents — are the essence of the motor system's task. We shall see throughout this thesis a wide range of body representations and control structures, indeed, one of the contributions of this work is a demonstration of this range. Almost all of these bodies and control structures have been implemented within a graph based motor system framework that we will call here the pose-graph. This structure was first constructed in 2000 and, although it has been re-implemented twice since then, its generality allowed the idea to survive intact through collaborative and solo work over the last four years. A brief review of the original contribution is supplied here for completeness and to ground the subsequent recent extensions and revisions to the framework.

A pose is a fundamental atom of a pose-graph motor system. In a representational creature it might be equivalent to a keyframe of an animation — a complete description of the position of a classical, hierarchical, joint-angle based digital figure. This dealing in terms of poses rather than key-frames, animation tracks, or joint angles is the first abstraction that the pose-graph structure offers. Additionally, poses may be parameterized — for example, a single pose might represent a key-frame bundle of a frame of a walk-animation that is itself re-computed based on blending several walk-animations together.

In the pose-graph motor system, poses, as you might expect, are arranged in graphs — specifically graphs that are directed and strictly cyclic (where all nodes are accessible from all others). Paths through these graphs are, in the case of a classical digital figure, the raw material from which animations can be created. In order to build an animation from a path through a graph of poses we need one additional piece of information — the durations between the key-frames. We call this the time-metric.

This graph structure becomes useful for two main reasons: because there are well-known algorithms for manipulating and inspecting graphs (in particular searching them, but also comparing them and editing them) and because we can specify operations at the level of the abstract-pose or chain of poses rather than at the joint level — in the simplest case, at any given time the configuration of the body of the creature is completely specified as either being at a pose graph vertex or at some point along a pose-graph edge. These two abilities allow some efficient abstract reasoning about and manipulation of poses to be constructed without committing to a particular pose representation.

For example, the standard search algorithm used for generating animations from pose-graph poses is the A*-search, for similar uses: **J.C. Latombe**, *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

A pose-graph motor system spends much of its time producing animation material by searching for poses from the graph that match certain criteria, then searching for paths to those poses from the current body configuration and then finally building "interpolators" that execute that path. To search for a path between two nodes in a graph we need another metric (which might be related to time, but is more likely related to an estimation of "effort") which we shall call the distance-metric. Two related distance metrics are possible (and in general needed for fast searching) — metrics between pose-graph nodes that share an edge and a distance between any two poses. To search for nodes of interest in the graph we need labels and other information stored in the graph. Sometimes these labels are simply names of poses of particular interest ("sitting" for example), sometimes these labels are the positions of part of the creature's body — the dog's nose. This decomposition of searching into two unrelated and relatively efficient searches — a search for a node, and a search for a path — works well for a number of problems.

both these metrics are used in the A*-search algorithm that the pose-graph uses to find shortest paths between nodes; it is a sufficient condition for the admissibility of the pose-pose metric as a search heuristic for the node-node metric for the node-node distances to never be less than the pose-pose of their respective poses.

Thus the pose-graph motor system seems to have something to offer the problem of *planning* and *sequencing* movement and its graph structure is one representation of the *constraints* on motion placed on a body by the availability of

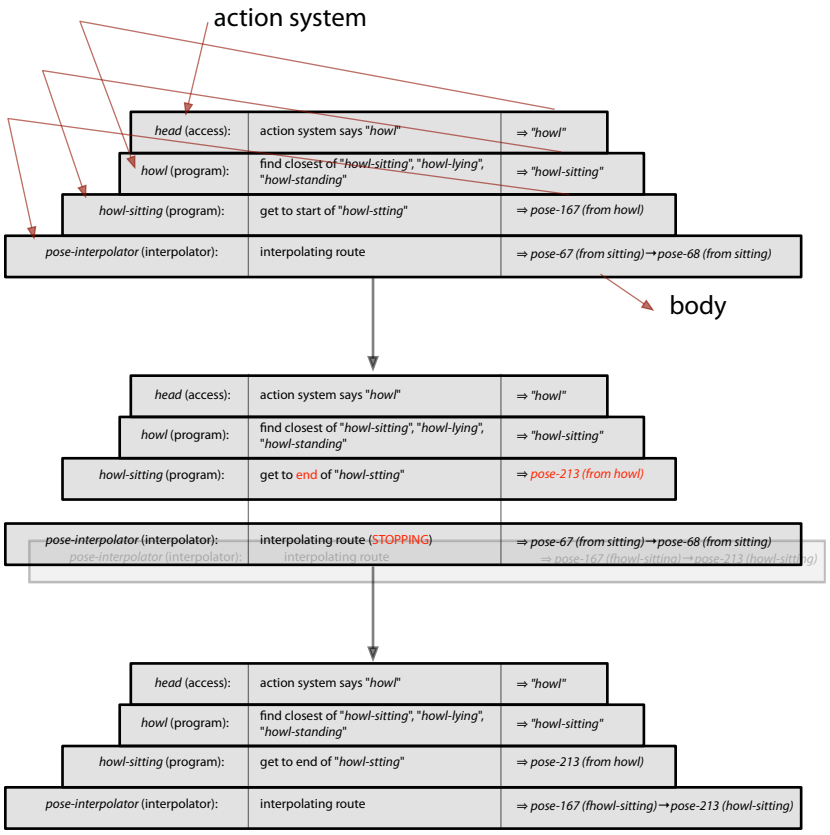

figure 7. Initially, the stack consists of a "head" element that is connected to the action system. Each program executes and requests that the pose-graph move to a particular object — which might be another program (in which case this is placed on the stack beneath) or a pose (in which case an interpolator is created, and the stack ends). The stack of executing motor programs undergoes a complex tear-down process whenever an element of the stack changes its output.

animation material *content*. To complete the basic pose-graph framework, a few more ingredients are required.

Firstly there needs to be a **control language**, situated between the desires of an action system and the constraints of the pose interpolators. One particular, "stack-based" language certainly found some amount of traction in the collaborative works presented here. This virtual machine for motor programs consisted of a running stack of control elements — the"output" of element n is resolvable into a request for a particular element $n+1$. Should element n ask for something and never change its mind, element $n+1$ continues to execute. Should element n ask for something else, the stack elements $> n$ are torn down, and a new set of stack elements are recursively constructed. Stack element 0 is a connection to the requests of an action system and at the top of the stack, element N, is a pose-graph interpolator. This construction worked well for a wide range of creatures — allowing the creation of modular, reusable, programs that shielded the action system from the current contents of the motor system — but began to show its age and its simplicities with more complex configurations.

In particular, for most creatures the identity between a single pose edge in a single pose-graph and a complete description of the creature's body is just too simple. Most representational creatures require the creation of several pose-graphs, which execute in parallel and in general control various layers of the body's motion. There might be a"tail layer" or a"face layer". The description of a body configuration is then, at the very least, the pose-graph edge positions of all of the pose-graphs, the parameters to the poses either side of those edges and some description of how the independent pose-graph layers blend. The analytic power of the pose-graph is not lost if, for example, there is one main pose-graph that controls the bulk motion of the creature, if there is one place to go to ask if the creature is sitting or standing. But the expressive power of the stack-based

motor programs are compromised if there are constraints between various layers. One critique and replacement is offered for *The Music Creatures, page 136*, and for agents where the decomposition into layers offers insufficient granularity, where motor programs that access different resources and target different bodily degrees of freedom need to run in parallel rather than series, this, more sophisticated motor-program "language" seems vital. The agents of *how long...* operate in this domain.

Secondly, we need some algorithms that can supply some analytic, or **perceptual representations** of what the pose-graph is doing. One is offered in *Dobie* — an algorithm that can construct a data-driven model from several, multiple pose-paths. This means that *Dobie* can learn that a particular flow through the graph, one that might have no specific motor-program to create it, is important enough to be named and represented as a possible action in the action-system.

Another is to use the pose-graph as a place to store learning information that can later be recalled in order to provide an analytic view of the motor system. This approach has many uses and the pose-graph, in the agents that use it, seems an ideal representation in which to store information about the relationship between motion and something else. One satisfying result is in the integrated learning of *Dobie* where, by incrementally labeling poses with the actions that are running when they are executed, an agent can reflect upon the appearance of its own body — in effect, averaging over the complexities of motor-program stacks that interface between action and movement. Such a capacity is one of the differences between a machine learning problem and a creature-*training* problem — how the agent *appears* is all a trainer gets to see. In this case it's extremely important that the trainer's reward information is propagated to the correct actions — not to the actions that are currently executing but rather to the actions that are generally responsible for making the creature look that

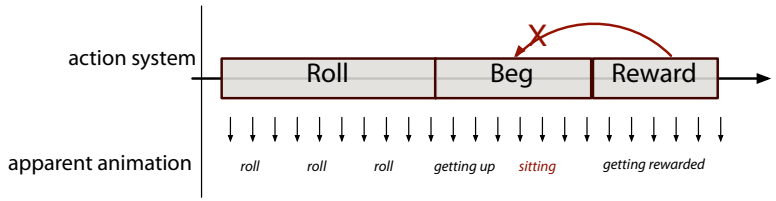

figure 8. The body does not always appear to be the same as the running action. Fundamental to the problem of building motor systems is having them be able to provide information about how the body *appears* to the outside. In the case illustrated above, the action system tells the motor system to perform the action "beg" — however, *Dobie*, in transitioning from "roll over" to "beg" appears to be sitting for a moment. It is the action that causes "sitting" that ought to be rewarded.

way. A trainer rewarding *Dobie* the virtual dog for sitting is rewarding the appearance of sitting — and the fact that a complex stack of motor programs is at this instant preparing to make *Dobie* stand is hidden and thus meaningless as far as the trainer is concerned.

We can store this information in each pose as a simple distribution $p(\text{action})$ and we can compute the entropy of this distribution which indicates whether this is a pose that is strongly associated with the particular action or not. We can improve these histograms, decreasing their sensitivity to precise system timings, by smoothing these labels along the graph edges.

In *The Music Creatures* we have examples of storing sound in the pose-graph — the *line* agent, *page 125* — and of learning the results of more control-like poses — in the *exchange* agent, *page 119*.

The usefulness of storing this information, and propagating it around the graph structure hints at a more general extension of the pose-graph into hierarchical structures, that allow the storing of label information, and perhaps even pose information at a variety of resolutions. Indeed, as the richness of the information stored in the poses increase, so does the chance that our search problems will not decompose into node-finding and then path finding. Searching for a node in the graph that has certain apparent properties that are dependent on the path that the agent would have to take to get there. To find these kinds of nodes takes, at the very least a full $O(n^2)$ search of the graph (a full, carefully ordered, breadth first search) with $O(n^2)$ storage — although the A*-search algorithm also scales as $O(n^2)$ its effective leading constant is much, much smaller given the good pose-pose heuristic information. Hierarchical graphs support a downward, heuristic beam search that reduce this search to $O(n\log(n))$.

A review of hierarchical search methods in such ad hoc generated hierarchies is: **J-A. Fernández-Madrigal and J. González**, *Multihierarchical Graph Search*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24 (1), January 2002.

For a use of the hierarchical motor system work described here:

C. L. Breazeal, D. Buchsbaum, J. Gray, D. Gatenby, B. Blumberg, *Learning From and About Others: Towards Using Imitation to Bootstrap the Social Understanding of Others by Robots*. Artificial Life, 11 (1), 2005.

There are a variety of ways of creating low-resolution versions of connected graphs. Those most useful for searching maintain the same connectivity information at lower-resolution views as found in higher-resolution views, looking at the connectivity of nodes as well as their contents when deciding to merge nodes. Since these graphs are completely cyclic anyway, this is not a hard constraint, but a heuristic. We should expect to find that the higher-resolution children of two adjacent low-resolution nodes are also reasonable closely connected. In the hierarchical pose-graph we collapse chains of singly connected nodes — common in graphs created from long animations — into fewer nodes by unsupervised clustering while converting nodes that are centers of star configurations into larger models centered on them. At any given time, the agent's body is located now not only on a pose-graph edge or node but on a whole set of increasingly lower-resolution pose-graph edges or nodes. Poses that fit some description and are "close-by" in terms of the shortest path from the current position can now be found with standard hierarchical graph searching techniques. These techniques are used in the motor learning of the *exchange* creature in *The Music Creatures, page 125*, and also form the basis for some of the motion identification work described.

The generic *pose-graph*

It is worthwhile to stand back and look at the more general implications of pose-graph-based motor systems in a more digital-art rather than digital-agent context. The triple space navigated by the motor system — action selection, virtual body and animation material — is particularly interesting. The pose-graph was originally constructed as a structure that could take *content* in the most banal sense of the word — unstructured animation produced by hired animators — and reuse it, live. This repurposes and exploits the animations, the animator's talents and the extremely polished and researched tools that animators use. As a site of content the pose-graph explicit representation of material

seems inherently collaborative. In *Loops* the pose-graph structure became the representation used to manipulate not *movement* (the underlying motion used in that work simply played out on a loop) but *choices* — rendering parameters and action-system parameters named by the collaborators. The explicit graph structure — its graph of statements that "this" is a pose and it has "this" name — became the sketchpad for the solidification of the collaboration, the memory of where we were. As such it became a site of versioning information and database techniques, *page 91* and later, *page 209*.

3. Critiquing c5, *alphaWolf*

AlphaWolf was the vision of Bill Tomlinson and is described in detail in: **B. Tomlinson**, *Synthetic Social Relationships for Computational Entities*. PhD Thesis, MIT, June 2002.

The c5 action-selection mechanism is an example of an organizing structure rather than a prescription for action selection in general. Each "slot" in this template-like description can be filled by code of incremental complexity; the abstraction afforded by c5's action groups has survived for a number years and a number of projects because it offers the right blend of structure and openness for a number of problems.

Much of this approach's gentle innovativeness was devoted to controlling the temporal patterning of the resulting action selections. But it is exactly here that there was more work to be done. The overview of the c5 action-selection mechanism given above was an overview because it omitted two complications. The full algorithm, as ultimately deployed in *alphaWolf* and *Dobie* (for *Loops* the difference is immaterial) is given here →

Note the three highlighted lines in the description of the action selection mechanism. These, rather explicitly, avoid a common but troubling scenario — when the do-while of an action-tuple goes to zero, but the trigger of an action-tuple is non-zero, the action-selection mechanism is likely to dither (swap back

Modified C5 Action Selection Algorithm

state

startles — a list of action-tuples that get special privilege to interrupt others.

tuples — a list of action-tuples inside this group

currentlyActive — the currently active tuple

lastValues — a mapping from tuple to real number

| define expectedValueOf(*tuple*) to be *tuple*.trigger()**tuple*.value() if tuple is *currentlyActive* or *tuple*.doWhile()**tuple*.value() otherwise.

algorithm

if the maximum expectedValueOf(...) over all of startles is greater than zero then the greatest becomes *nextActive*.

| otherwise, if the *currentlyActive* is a startle, and expectedValueOf(*currentlyActive*)=0, and *tuple*.trigger()**tuple*.value() is greater than zero, *currentlyActive* remains active and *nextActive=currentlyActive*.

▸ otherwise,

construct the new map *nextValues*[*tuple*] = expectedValueOf(*tuple*) for all *tuples*

if any tuple that isn't *currentlyActive* has *nextValue*[*tuple*]>*lastValues*[*tuple*] and 2* *nextValue*[*tuple*] > expectedValueOf(*currentlyActive*) then select a new action

if expectedValueOf(*currentlyActive*)=0 then select a new action

▸ if we need to select a new action:

if all of *nextValues*[...] = 0 then nothing is done, and *nextActive* = *currentlyActive*

▸ otherwise,

sample *nextActive* from a normalized version of *nextValues*[...]

| if we selected because expectedValueOf(*currentlyActive*) was 0 set *nextValues*[*currentlyActive*] = *tuple*.trigger()**tuple*.value()

◂ finally, *lastValues*[...] = *nextValues*[...]

and forth) for exactly one iteration. Why? because the action-tuple appears to the action selection mechanism to go from zero to a non-zero value in one iteration, this causes a "surprise" based reselect. The first line prevents this phenomenon from occurring in the case of startle action-tuples, the second in the case of normal action-tuples. In order for these "workarounds" to be written at the action-group level the action-tuple must present all the pieces of its expected value function, coupling its internal value structure directly to the group. (There is an inverse situation, which was always solved at the tuple level, where a high trigger leads to a zero do-while, that again causes a one-iteration dither.)

However, these corrections do not remove all temporal pathologies from the action group formulation. If an action's trigger depends on the currently running action of the group, or if it depends on another action in another group which in turn connects back to this action, multi-iteration dithers and even freezes are possible. Such cases have appeared a number of times — during the development of *alphaWolf*, where the creatures' action systems were split into two separate, but very coupled groups. This coupling is perfectly reasonable on paper — sometimes, the "attention" group controlled the palette of options for the the main "action" group, sometimes the what the wolf was doing controlled what it could pay attention to.

Other aspects of the extant c43/c5 creatures are already pushing along this path of temporal complexity. Traditional reinforcement learning systems propagate value between state-action pairings at each state-action transition. We have seen that c43/c5 potentially drops this credit-assignment stage altogether should the upcoming value be expected. However, in agents with complex bodies and motor systems this credit-assignment stage is further complicated. It is necessary (in the sense that it is often the difference between a "trainable" character and a non-trainable one) for creatures with complex bodies to defer this decision to propagate value to a later point in time. Since the action system necessarily runs

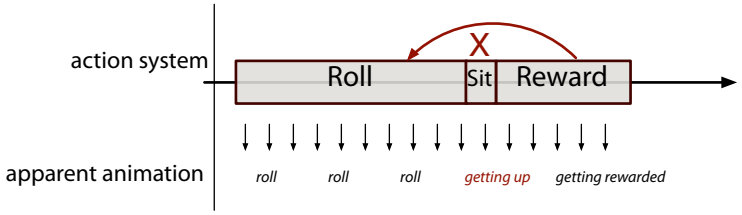

figure 9.Sometimes actions should not participate in credit assignment. In this case the action"sit" is too short to have an effect on the body of the character — "Roll" should receive the reward instead.

ahead of the motor system, issuing commands that will take a number of frames to have an eventual effect on the body (due to animation constraints and the constraints of realistic motion) a creature must take care to allow only actions that had a shaping influence on the body of the character to participate in the value propagation.

The reason for this descriptive detail (in addition to the benefits of accurately recording the action selection algorithm for *alphaWolf*, *Dobie*, *Loops* and an early music creature for the first time) is to illustrate where the tensions lie in taking the simple intuition behind the c43 action selection mechanism and turning it into an algorithm. The general lesson: that unexpectedly difficult intricacies involving the patterning of time result from surprisingly simple systems. The specific lessons: that if one really thinks of the action selection problem in terms of not only relevancy and persistence but *patterning* — the order and timing of actions that arise from an action system — then c5 appears deficient. As we move from what one would call patterning, towards actions systems that distribute and control multiple interacting actions simultaneously, what I might be tempted to call *choreographing* actions, then one might need an additional level of description, additional state and additional control mechanisms than those offered by c5's action-group.

Locating c5

Before proceeding to the artworks created with c5, it's worth pausing to locate c5 in the space of possible action-selection solutions, considering in more general terms the problems that it solves well and the kinds of problems it has little to offer. We'll reuse the axes later to work out where some of the contributions made in this thesis fit. These axes are judged from the point of view of the author of an agent, when they are trying to think through or think over the behavior of the system. For the purposes of comparison we'll also include the popular graphical environment Max, which will be the subject of considerable discussion in the last chapter; and a hypothetical placeholder entitled xFSM that notates a variety of small, finite-state machine techniques that appear in scripting contexts or simple computer games. Finally, Maes89 refers to Maes's influential "How to do the right thing" architecture; and Scoot, an older, hierarchical Synthetic Characters Group architecture.

P. Maes, ***How to do the right thing***, MIT AI Lab, Memo 1180, December 1998.

Scoot: **S-Y. Yoon, B. Blumberg, G. Schneider,** *Motivation Driven Learning for Interactive Synthetic Characters*. Proceedings of 4th International Conference on Autonomous Agents, 2000.

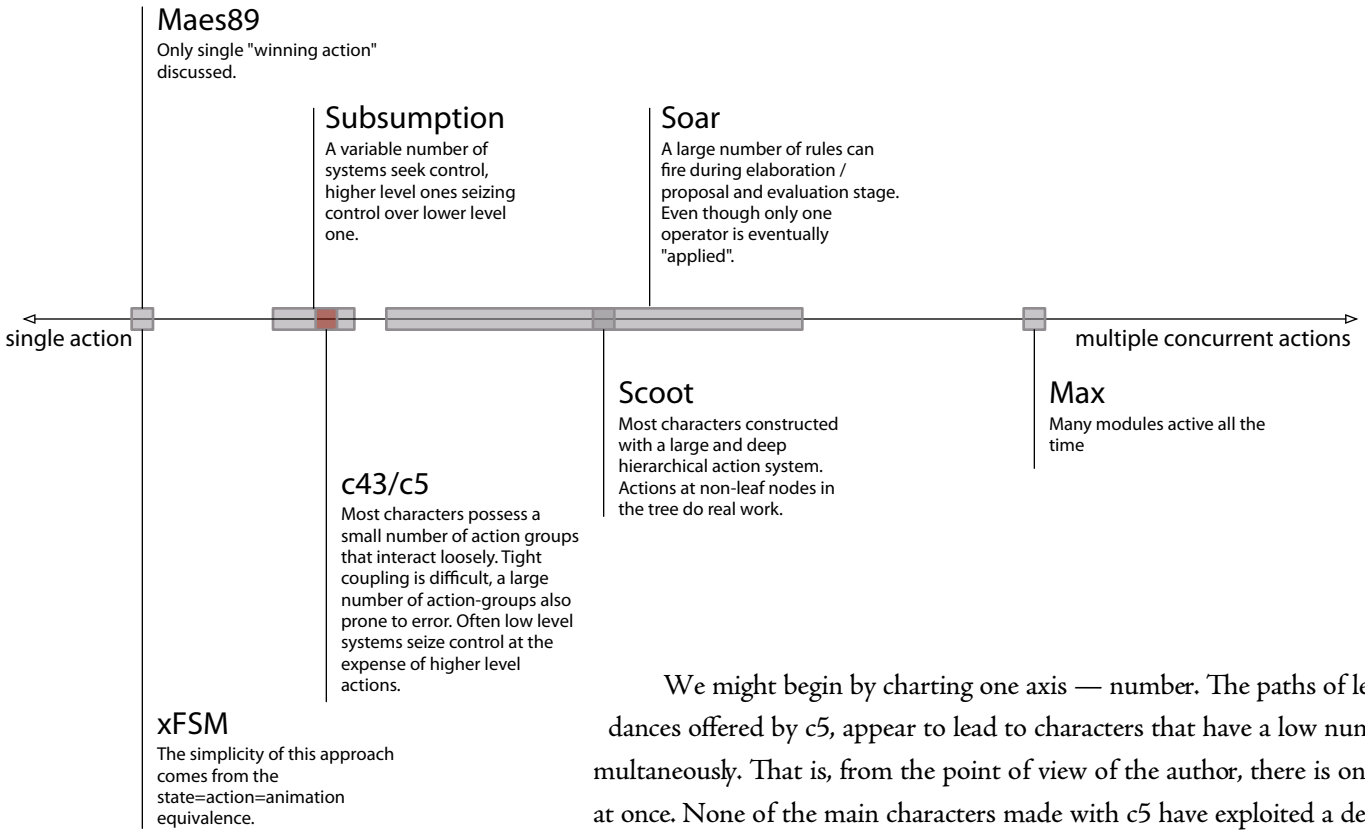

figure 10.
Single-actions versus multiple actions.

We might begin by charting one axis — number. The paths of least resistance, the affordances offered by c5, appear to lead to characters that have a low number of actions active simultaneously. That is, from the point of view of the author, there is only a few things going on at once. None of the main characters made with c5 have exploited a deeply hierarchical action-system, and this is probably because c5 says little about, and potentially has trouble over, action-tuples that interact with each other "behind the scenes". Towards the opposite end of this axis we locate hierarchical action structures (for example, Scoot) and other structures where the composition of simultaneous actions and finally the composition of simultaneous, interacting actions leads to the expressive power of the action-group. As we proceed through this thesis we will move from left to right — as we meet, as artists, the duration of a dance, or the installation of an artwork, we will require more complex, more self-generating temporal structures based on multiple overlapping actions.

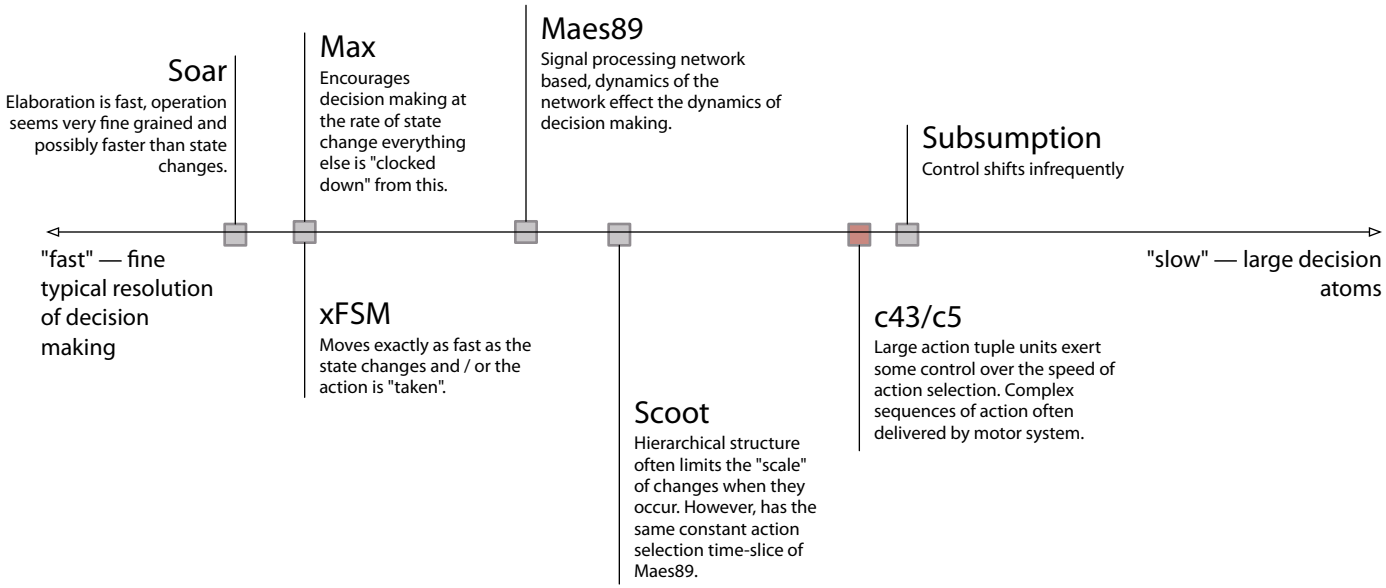

figure 11.
High resolution versus low resolution action selection.

Another axis — **speed**. What is the typical granularity of decision-making inside the action system that the author is actually working with (or against)? Systems that make decisions based on elaborate filter networks count here as fast; Planning-based systems must potentially discard a large amount of computation to change their decision count as slow. C5, as we have seen, picks a middle ground, selecting infrequently and allowing both active actions and inactive actions to semi-explicitly recommend re-selection. As we proceed through this thesis we will fight to maintain the correct granularity of decision-making and augment c5 with both long and shorter time-structures.

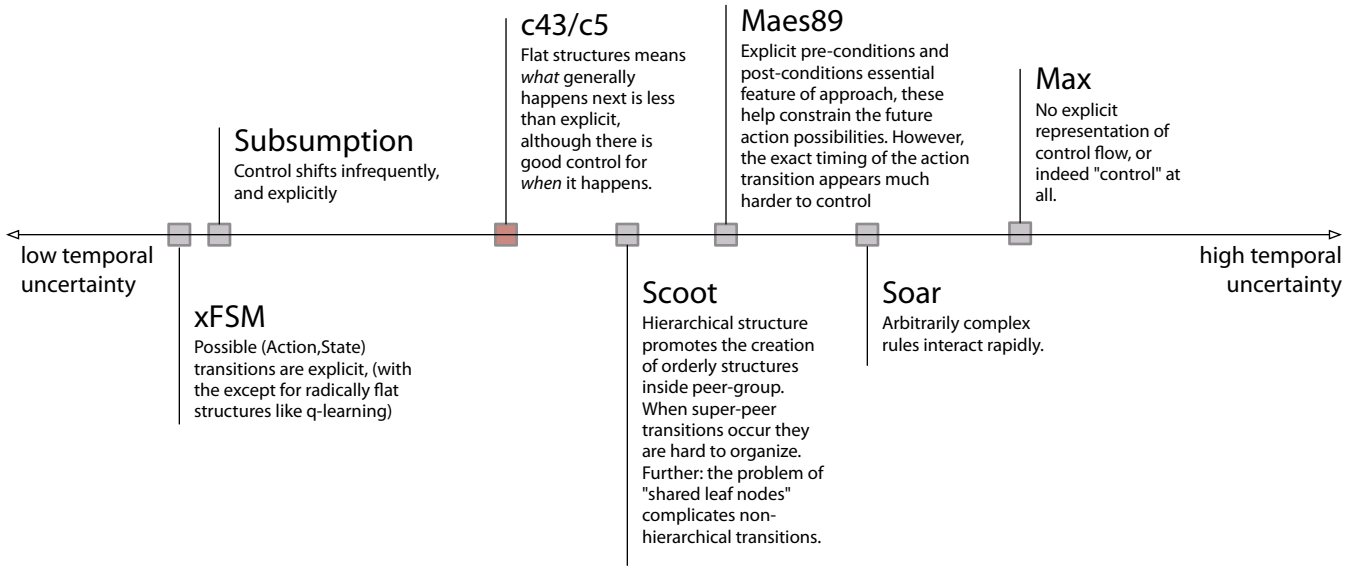

figure 12.
Low versus high "temporal uncertainty". What is the range of things that can happen when?

The final axis — **temporal uncertainty**. What is the typical potential temporal complexity of the results of action selection — both its "texture" in time and its scope? Here scripting systems and finite-state machines count as low — these are often the kinds of systems that are used in computer games particularly because they are considered simple, and "safe" to use. C5 is significantly further along than this, with its broad, flat action-groups (that have a lot of scope) and its fairly unconstrained transition dynamics. Without taking extra steps patterning sequences of c5 action-tuples is left entirely to the author, often delegated to the motor system which has significantly stronger language for talking about constraint and ordering. I will describe ways of simultaneously increasing the temporal complexity of agents and their action selection techniques while reducing this "temporal uncertainty" from the point of view of their authors.

Some of the work of the latter half of this thesis is to broaden the range of these axes that is accessible from c5-like systems. Some artworks — *Loops* and *Loops Score* — will need more actions ongoing simultaneously and interacting with each other, and I'll present two ways of achieving access to this part of that axis. Before moving onto those artworks, there is an installation that will draw the advantages and disadvantages of c5 into sharper focus.

alphaWolf, *a large c5 installation*

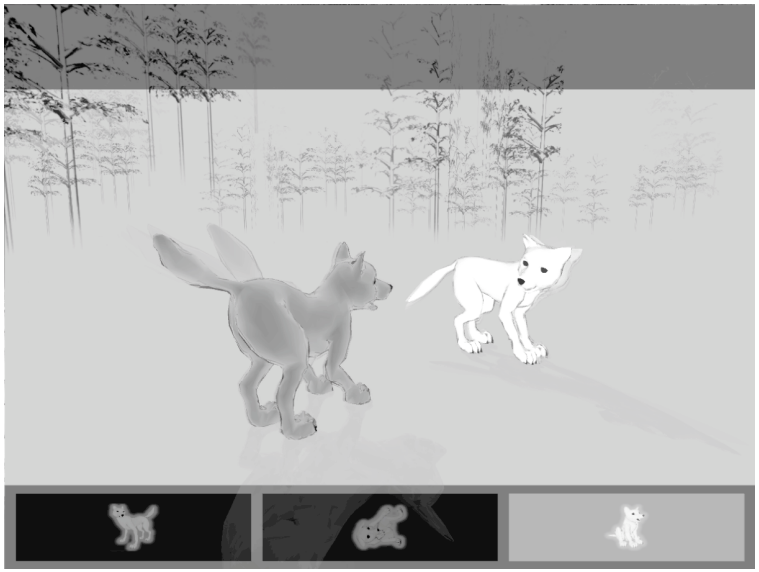

figure 13. *alphaWolf*, 2001, The Synthetic Characters Group.

alphaWolf was a project by the Synthetic Characters Group during the year 2001, completed during an intense collaborative period during the summer, premiering at the Siggraph computer graphics exhibition. It is by any metric a success, multi-participant interactive work. It later toured to a number of venues, including the international Ars Electronica festival and German Zkm institute. Since in terms of the number of people actively contributing code, and the numerical size of the perception, action and motor systems it contains, it is second only to *how long...* it is important to take this opportunity to examine a concrete example of a complex work created within the c5 agent toolkit and to learn the lessons latent in this unquestionably successful installation.

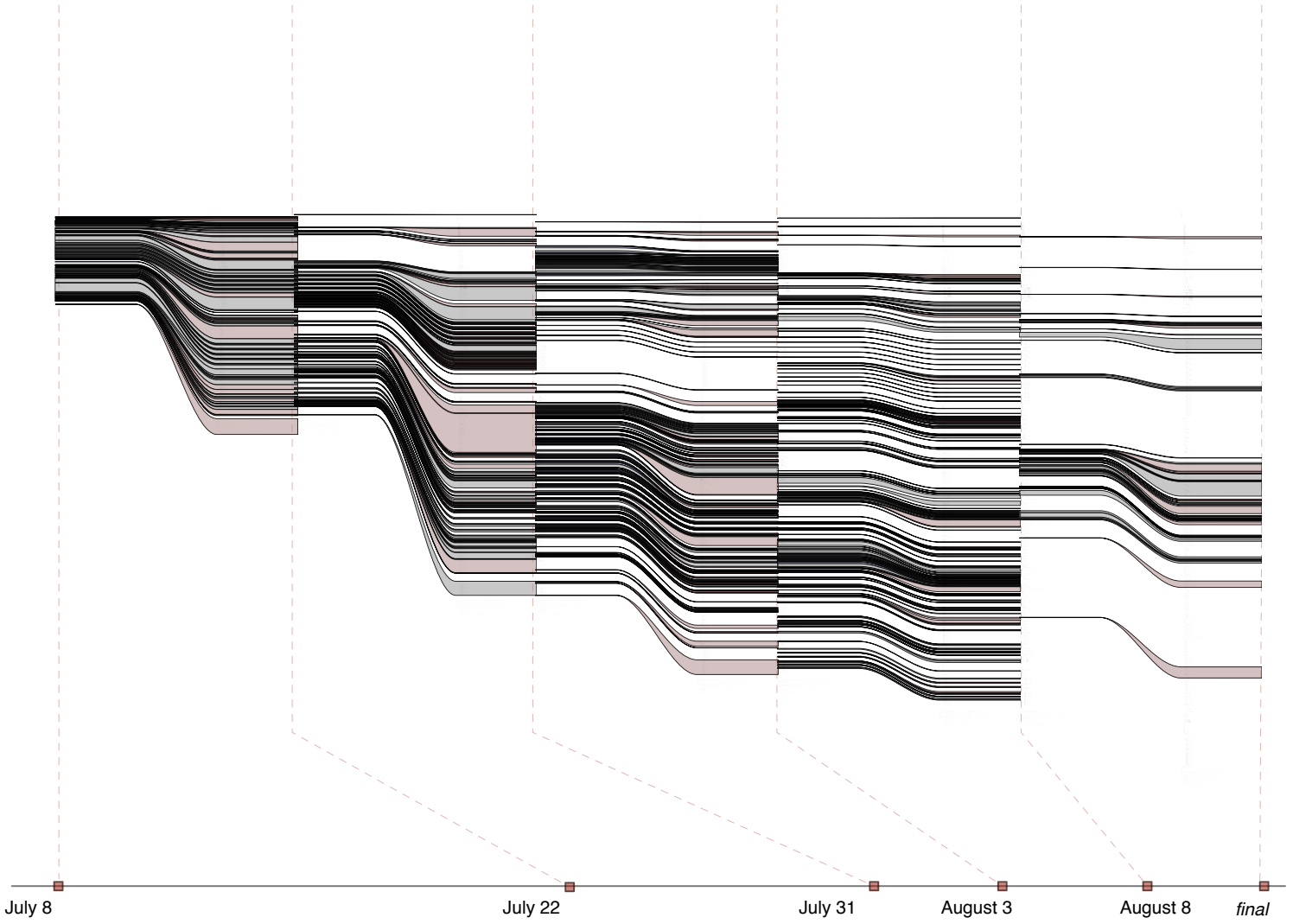

figure 14. The main action system for a wolf in alphaWolf was one of the main sites of authorship during the development of the piece, and grew seemingly without bound. Changes (gray) and additions (red) continue to be distributed throughout the file even up until the last day.

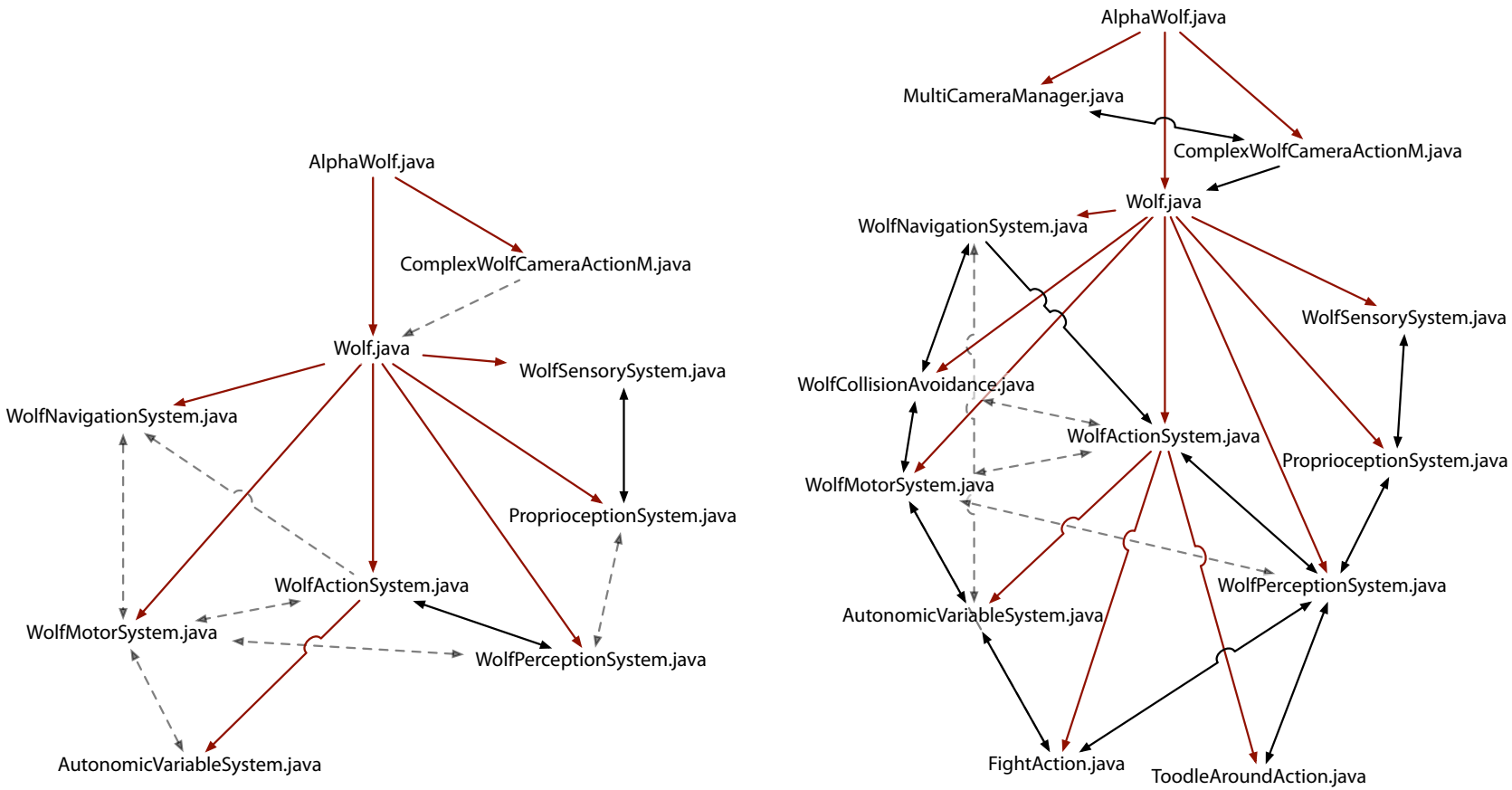

figure 15. Over time, everything becomes connected, and coupled, to everything else. The left figure, taken from early July shows how the systems are instantiated (red), directly connected (black) or weakly coupled through the wolf working memory (dashed, grey). Over time, however, these"abstraction barriers" crumble and a densely connected set of codependent files result.

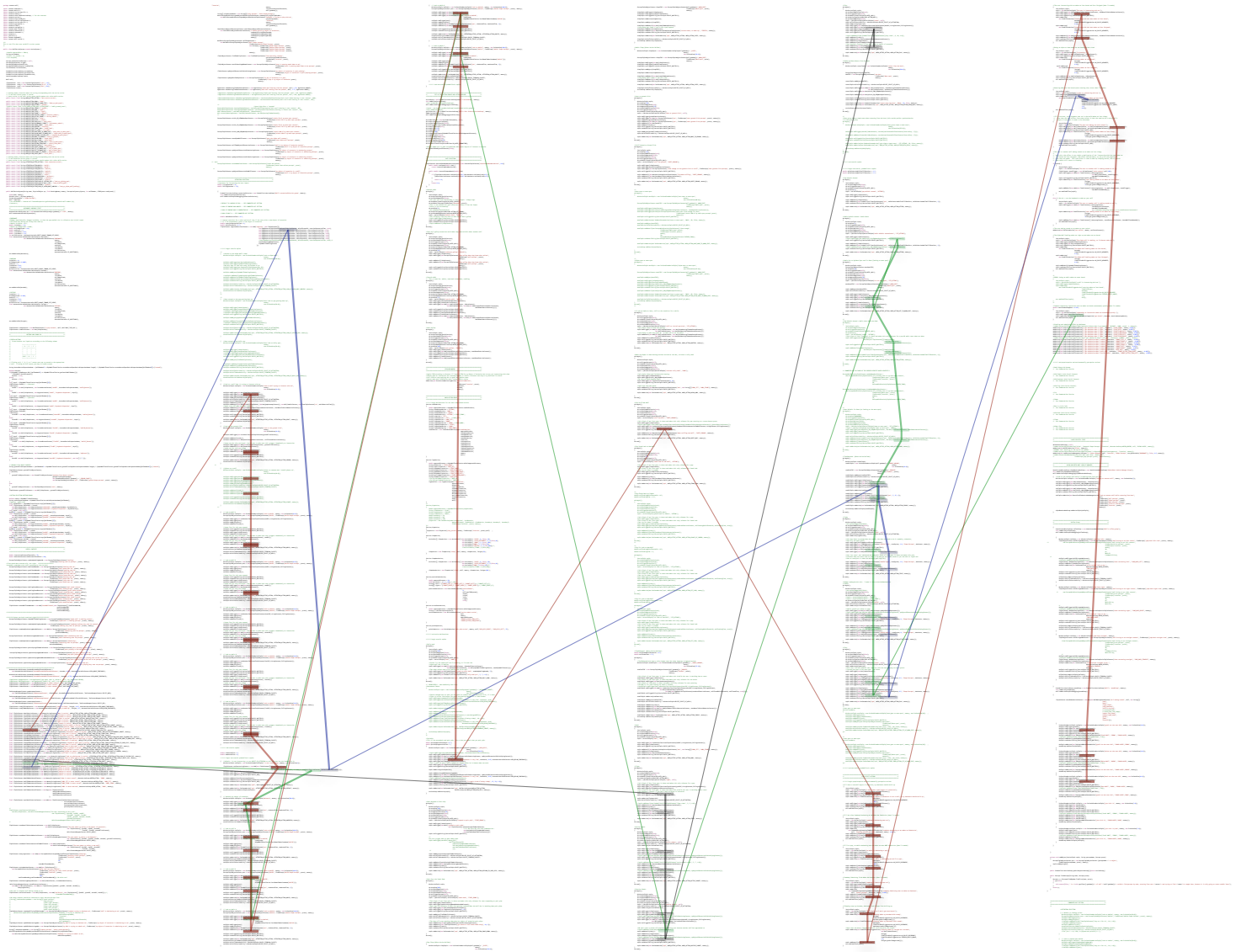

figure 16. Inside the main action system file independence between action-tuples' activations is impossible to maintain. This figure traces the thread of four"contexts" for triggering action-tuples that are themselves based on which action-tuples are active. Note how the colors weave in many different combinations — no single hierarchical reorganization of this file will capture the complex temporal patternings that these threads are producing.

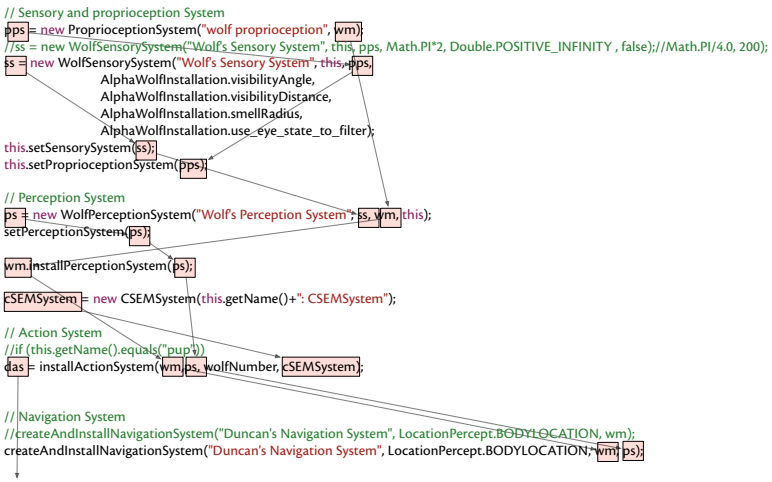

figure 17. The above, annotated segment, comes from the main Wolf character's constructor. Note the complex, coupled, and order dependent, initialization of subsystems.

By a "close reading" of the archives of this project I am tempted to find the following areas as problematic for the creation of complex agents in general.

Initialization, connection, registration and notification. All of the lines of Alphawolf.java (the main installation file) and Wolf.java (the main initialization file for a Wolf creature) that do not create a system are devoted to registering systems with each other and passing references to systems through constructors so that they can perform their own registration and connections. Early versions of *The Music Creatures* — as they cut new paths away from the toolkit, were similarly plagued by critically order-dependent but otherwise boilerplate instantiation code. This is the creation-time coupling issue, and in more complex and heterogeneous creatures there is a tenancy for this problem to increase in severity. Indeed, in *alphaWolf*, the burden of reconfiguring and disconnecting a creature is such that no creature is ever deleted from the work, at the end of a 5-minute interaction cycle, with the pups having "grown" into adults, the creature is reused rather than recreated.

→ I develop the Context Tree and a set of associated techniques to combat this problem.

No re-use or extensibility. The initialization of the wolf agents in the main behavior file WolfActionSystem.java is almost completely monolithic — it is inconceivable that as a description of behavior this class could be extended (or, in the language of object-oriented programming, sub-classed) by another to create a variety of wolf or a wolf with a particular behavioral style. Evidence for this hypothesis is present in the very form of the file, which includes in one overlapping place the descriptions for the wolf pups, the wolf "aunt" and the alpha and beta adults. Yet, at the same time, this complex initialization routine seems have little space left for yet more parameters and options.

→ I develop an extensible programming model based in the Context Tree that approaches the problem of creating extensible complex assemblages of systems with "over-ridable" or "sub-classable" default behavior.

Chains of actions are hard (B happens after A finishes), **deferred execution is hard** (B doesn't happen while A is still ongoing) — These two temporal primitives are so hard to express inside the body of the main agent behavior file that even a fairly close reading of the code will not yield this secret issue. Only by executing and debugging the code will one realize the careful choreography of signaling between one action-tuple and another, between one mode and another. The chaining of actions hides a subtler problem: Action-tuples whose values are based on the current activation of another action short-circuit some aspects of the action-selection mechanism, and reveal the complexities that the action selection technique was constructed to hide. Part of the issue stems from the occasional pathological cases in the action selection framework, some of which are described above *page 71*, but much of the problem is that there simply isn't any framework support for such chains of actions.

→ The Diagram system is constructed around making explicit the temporal sequence of events, incorporating some of the techniques that motor systems use to handle their temporal sequencing, and the Fluid tool renders the authorship of these sequences very visual while retaining the expressive power of programming languages.

That **execution ordering** is significant and difficult is also hidden inside the action-system file. Here WolfActionSystem.java goes to some lengths to ensure accurate hand-off between actions is reflected in the creatures "working-memory". Should a working memory slot go unfilled, or perhaps un-overwritten for one execution cycle, there are places where the action

figure 18. This figure shows the"optional" sections of the main action system file — areas activated in the cases that the wolf in question is the"alpha", "beta" or "aunt" wolves rather than an interactive wolf pup. Note also the fine scattering of numerical constants that control the behavior, one might even say the"character" of this particular character

system stalls and the behavior of the creature breaks down. This suggests that there is a disconnect between the time-scale that the actions are operating on and the amount of persistence that their effects should have.

→ I develop the generic radial-basis channel as a technique *page 119*, borrowed from the problem domain of motor systems, as a way of allowing actions to have short term effects that extend past their activation stories in a flexible way, diffusing the coupling between action and life-cycle.

Insufficient modularity of systems — things which on paper seem like they should be modules couple so much information into the main behavior class that they are added inline rather than as a module. We can see this in the perception-system elements, user-interface elements (the on-screen buttons for participants to interact with) or the action-system manifestations of device controllers. Other systems which are specified elsewhere (for example the perception system) have much of their structure duplicated as they too are coupled in (for example in the list of source action-tuple triggers). Such a static duplication of structure is unmaintainable in a creature that undergoes structural learning, for example *Dobie* overcomes this problem in a specific case.

→ We shall see the use of a Context-Tree-like structure to provide a easy integration and de-integration of systems, *pages 215*; generic dynamically extensible views of hierarchical structures which will form the basis for the Fluid environment, *page 377*.

Insufficient modularity of behavior — Even from the overview of successive differences between versions of the main action system file of *alphaWolf*, we can sense large blocks of code being introduced, scattered into the middle of the main initialization method. This impression turns out to be accurate upon closer analysis — very little of this action system was

figure 19. This figure illustrates a particular kind of "comment" inside the main action system file of alphaWolf. While comments are in general used to pass explanations between collaborations inside the code itself, the highlighted comments do not *explain* what surrounding code does (as typical comments do), but rather store what the surrounding code *used to be*.

tested or even is even testable in isolation. There is overwhelming support for isolated unit tests in the software engineering of complex systems. Why was it not attempted in this work? Some of reasons are plain to see in the relationship between the added blocks of code and their surroundings — they require too much from their surroundings to be efficiently tested in isolation, they couple so strongly to their environment that to isolate them would require an accurate duplication their environment.

→ The lessons learned will form part of the motivation behind the Diagram system, where we explicitly decouple systems (for example"abstract balances", *page 252*) as well as some of the techniques used in *The Music Creatures* where the coupling between systems is the subject of long-term (that is longer than the execution cycle of a creature) learning, *page 127*. Finally, we design the *Fluid* environment to promote a bottom-up testing methodology by incorporating techniques that allow small components to be assembled into longer sequences — effectively allowing the reuse of the small testing scenarios.

Storage of history — Finally, we note the growing number of comments, notes and markers in the code, the annotation of and the incorporation of the history of the file into the file itself — as notes to collaborators or to self. Is this the correct place for it? And what history is missing — certainly the expected execution orderings that are implicit in the file, but also the to-ing and fro-ing of the numerical parameters in the file as the behavior is pushed one way, perhaps by one collaborator, and another, perhaps by the addition of a new behavior.

→ I develop an environment, Fluid, for creating, monitoring and debugging complex assemblages of code that explicitly offers more historical information about its own use, *page 390*. I also create specific, long term database structures for other aspects of a work and the work's agents,

page 127 and *page 209*.

There will never be another *alphaWolf*, and certainly not simply for the purpose of comparing a programming technique with another. The exciting work of 6 programming collaborators who remained dedicated to one vision of an installation for more than 3 months cannot be duplicated for the purposes of an attempted scientific comparison. So these ideas will be developed throughout this thesis and be tested in other installations, both bigger and smaller than *alphaWolf*, but we shall not be able to return to the exact same project for a conclusive demonstration of their applicability to this particular domain. However, it is worth noting that there was at the time and beyond consensus amongst the collaborators on *alphaWolf* that this work took us to a plateau of complexity that could not be safely crossed in a reasonable amount of time— neither with an extra pair of hands or an extra month.

It was with this thought that I turned to the next 4 years of work — developing the tools and techniques required to traverse this threshold as I took the agent, and the agent toolkit, into new territory.