Representational Complexity of Reactive Agents

Frederick W. P. Heckel, *Member, IEEE*, G. Michael Youngblood, *Member, IEEE*, and Nikhil S. Ketkar

***Abstract*— Reactive agents are an important part of video games and numerous tools have emerged to facilitate the rapid construction of such agents. While the ability of the commonly used reactive techniques to express agent specifications is roughly equivalent, the authorial burden of constructing these specifications varies. In practice, this means that identical agent behavior may be more difficult to create in some architectures than others. In this paper we introduce the notion of representational complexity that relates to the authorial burden of constructing such agents and theoretically compare the representational complexity of finite state machines, behavior trees, and subsumption architectures. Our key finding is that hierarchical subsumption architectures have significantly lower representational complexity as compared to hierarchical finite state machines and behavior trees, which makes subsumption the best choice when developing authoring tools for non-expert users.**

I. Introduction

Real-time games are a thriving form of entertainment that are slowly replacing television and movies as popular forms of entertainment [1]. Apart from entertainment, games have become indispensable in training simulations (e.g., ELECT BiLAT [2]). Games often play major roles in facilitating mission rehearsal, environment familiarization, and cultural awareness and are now important in training armed forces and security professionals.

One of the most important parts of many real-time training simulations is the creation of intelligent agents to model other actors in the scenario with whom the trainee must interact. This could include peers on a team, enemies in a battle, or victims at a disaster site—other individuals who would be present in the real situation. While much progress has been made in developing tools to model environments and interfaces for training simulations (such as level editors for 3D environments), the tools to model intelligent agents lags far behind. Often the design of intelligent agents requires programming, which may considerably slow development. This is potentially a serious problem if domain experts are needed to design realistic characters, as domain experts in military strategy may not be familiar with artificial intelligence development or programming.

Simple, reactive agents are frequently used in highly dynamic game environments, as the computational resources available to the AI may be too limited for large numbers of planning agents. Recent games have used planning for AI, but for numbers in the tens of agents; reactive systems can allow hundreds, if not thousands, of characters on consumer hardware [3]. An example of such agents would include bystanders at an accident (in a training simulation) or zombies in a computer game (for instance, Left4Dead) [4]. These agents have relatively simple specifications and thus there is an opportunity for building authoring tools that allow users to build intelligent agents without programming. Intuitive and effective authoring tools could not only speed up the process of building simple reactive agents, but also make it easier for domain experts to create deeper, more realistic characters. In addition, accessible tools for creating AI characters could potentially open a new area of user-generated content in which players not only create worlds and scripted game modifications, but also new characters which act in interesting ways to enhance the game experience.

The subject matter of this paper is the theoretical frameworks that form the basis of authorial tools for building simple, reactive agents. While there have been numerous tools for building such agents (tools based on finite state machines, behavior trees and subsumption architectures), the effectiveness of these frameworks from the perspective of authorial tools has not been analyzed. In this context it is important to make a distinction between the ability of a theoretical framework (or agent language) to express an agent specification and the ease of constructing such a specification. While alternative frameworks can have equivalent power of representation, the representational complexity of the agent specification may vary. The goal of this paper is to introduce metrics for measuring representational complexity and analyze a number of frameworks for building reactive agents. We begin by introducing the key frameworks used for building authorial tools and their variations.

II. Background

Many different reactive architectures for agent control exist, though only a few are commonly used for games. In general, reactive agents can be said to map from an input of *perceptions* to a set of output *actions*. Rather than relying on an internal model of the environment and producing plans, reactive agents make decisions based on the world state directly. For many robotic applications, the lack of state in reactive agents can be greatly limiting, but in games the environment is already a world model. This means that world state may contain symbolic information, rather than the low-level sensor readings that robots depend on—this allows more powerful controllers to be built with relatively simple architectures. Relaxing the limits on state for reactive controllers creates behavior-based controllers, which provide additional power [5]. For the sake of this work, we consider behavior-based variants of the evaluated architectures to be very similar, as the basic representations are the same.

Frederick W. P. Heckel, G. Michael Youngblood, and Nikhil S. Ketkar are with the University of North Carolina at Charlotte, 9201 University City Blvd, Charlotte, NC 28223-0001. Email: {fheckel, youngbld, nketkar}@uncc.edu

Finite state machines (FSMs) and hierarchical finite state machines (HFSMs) are a popular and simple to use method for building reactive agents. They are inherently modular, allowing a library of primitive behaviors to be developed along with transitions based on combinations of simple conditionals. The behaviors can then be connected by these transitions using a graphical representation. While FSM-based methods are simple as a concept, they suffer from the problem of becoming unmanageable past a certain size, as maintaining the $O(n^2)$ transitions between states becomes an increasingly difficult problem. HFSMs reduce this issue by providing super-states that encapsulate portions of the machine. Concurrency issues can be addressed with non-deterministic finite state machines by treating ϵ transitions as indications of states that run in parallel. Without this convention, concurrency can result in an exponential increase in the number of states.

Additional extensions to HFSMs are provided in the behavior tree (BT) formulation of Damian Isla and Chris Hecker [6], [7]. In behavior trees, the HFSM is represented using a tree structure. Internal nodes of the tree act as transitions between leaf nodes, which execute behaviors. These internal nodes provide conditional tests and implicitly define transitions between groups of nodes; for instance, a *sequence* decision node results in child nodes being visited in order without explicitly creating transitions between the children. The result is that transitions are only explicitly defined for parent-child relationships and not for sibling relationships. Behavior trees also allow *impulses*, which effectively reduce the number of common subgraphs. An example of an impulse would be to give a subgraph multiple priorities in a *prioritized-list* decision node; under most circumstances, subgraph A should be lower priority than subgraph B, but given a particular set of percepts P, A should be considered before B. While the resulting behavior tree controller is equivalent to a similar HFSM, these abstractions can potentially reduce authorial burden, but the resulting transition functions have a higher complexity. Behavior trees also have a larger set of concepts that must be understood by the AI designer prior to building agents. Mechanisms to handle concurrency also exist in other work, but we focus on the base definition in this paper. Many other extensions to behavior trees exist, and implementations differ greatly depending on the problem and game. While we primarily treat behavior trees as HFSMs because the representation is frequently similar, other methods are used for execution. The most popular type is similar to hierarchical task networks (HTNs), and uses a greedy search without lookahead to execute the behaviors [8], [9].

Another reactive control method that may be used is subsumption architecture. Subsumption was proposed by Rodney Brooks as a robot control architecture [10]. Behavior-based control extends subsumption by allowing more state and richer internal representations while keeping the core idea of executing several layers simultaneously to control an agent [5]. Agents are defined as a series of prioritized layers. Each layer is composed of a triggering condition, a primitive behavior to run, and a policy for interacting with other layers. Higher layers may adjust the control of lower layers, but lower layers may not interact with higher layers. Higher priority layers may completely override or *subsume* lower layers, or adjust their output in some way. Transitions need not be explicitly specified, though the closest analog in subsumption is the subsumption policy. Subsumption is inherently parallel, which eliminates the need for constructions such as ϵ transitions or schedulers. The prioritized structure of the layers reduces the number of subsumption policy decisions to be much smaller than the number of transitions in an FSM. Behaviors can be executed in sequence by composing a layer of multiple behaviors, but executing layers in a sequence is more complex than in a standard FSM unless each layer has a perceivable effect on world state. As with FSMs, a hierarchical variant of subsumption is possible. Hierarchical subsumption allows each layer to be composed of a subsumption layer stack, which can improve abstraction and modularity in the agent controller.

A. Behavior Editors

Behavior authoring tools have been developed both in academic research and in industry, and have been recognized as an important step towards improving artificial intelligence in video games during sessions at the 2008 Game Developers Conference, the 2009 Artificial Intelligence for Interactive Digital Entertainment Conference, and other events. A variety of tools exist from robotics, artificial intelligence, and the game industry. The RobotFlow interface from the University of Sherbrooke works with their MARIE framework to build complete robot software systems, but is more low-level than desired for game tools [11].

Artificial Technology, Xaitment, Havok, Presagis, and SimBionic have all produced commercial tools that allow the graphical creation of game agents [12], [13], [14], [15]. These tools are targeted at professional AI and character developers in industry, and are not generally appropriate for novice users. The commercial tools are included with full AI middleware and runtime systems, and are not intended to work with other AI middleware. Crytek includes the Sandbox editors with copies of games using their engine [16]. While Sandbox is available to end-users, it has a difficult learning curve. The Sandbox AI editor is FSM-based, but the terminology used in creating individual states is nonintuitive. All of these editors focus on providing authoring environments for finite state machine controlled and scripted agents. Various game studios have developed their own editors for internal use, such as Sony's Situation editor, developed by Brian Schwab for use in sports games [17]. A behavior tree editor has been developed for an upcoming release of the Visual3D Game Engine [18]. Additional behavior tree editors have been developed by the community [19]. To our knowledge, the only published user evaluation of an AI editor has been performed with UNC Charlotte's BehaviorShop, as seen in Figure 1 [20]. BehaviorShop is

Fig. 1. The BehaviorShop game AI behavior designer is a subsumption-based editor for creating controllers for game characters. The top image shows the subsumption editing screen, which allows the user to create new layers and rearrange existing layers through a drag and drop interface. The lower image shows the layer edit screen that is used to fill in the details, including the behaviors and triggering conditions, for individual layers.

a subsumption architecture-based editor targeted at a non-expert user base that has been very successful in allowing users to create characters even when they have very little or no AI experience (or programming experience).

III. Representational Complexity

Different agent architectures have different strengths and weaknesses. Qualitatively, we may talk about the difficulty of creating a specific behavior or type of behavior using each model. Quantifying this difficulty requires considering the complexity of the representation used to create, modify, and execute the agents. In this case, we are interested in the complexity of the graphical models used to create the agents, and need to quantify the number of components that a user must use in the creation process.

To demonstrate these differences, we present a scenario. While this scenario is very simple, it is possible to visualize each controller type and it sufficiently shows the difference in complexity of each architecture we are demonstrating. A simple scavenger agent lives in a world with three percepts: hunger, presence of food, and presence of a predator. This scavenger is only aware of three actions: search for food, eat food, and flee from predators. Its preternatural ability to

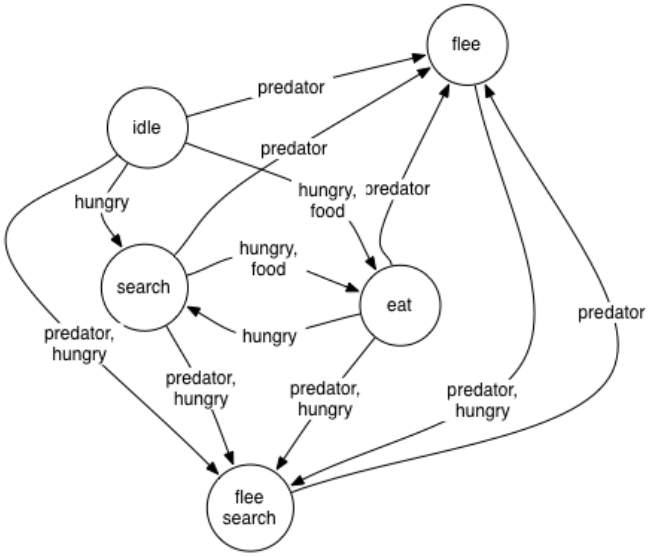

Fig. 2. Finite state machine representation of the scavenger agent. Note that only a few of the transitions are included for the sake of clarity. The total number of transitions, including those omitted, is 25.

sense food allows it to both flee from predators and search for food at the same time. A basic controller for this creature as a finite state machine can be seen in Figure 2. Since basic FSMs do not allow concurrency, five states are required. The creature can be idle, perform any of its three actions alone, or perform both search food and flee at the same time. This requires 25 transitions, as every state is accessible from every other state.

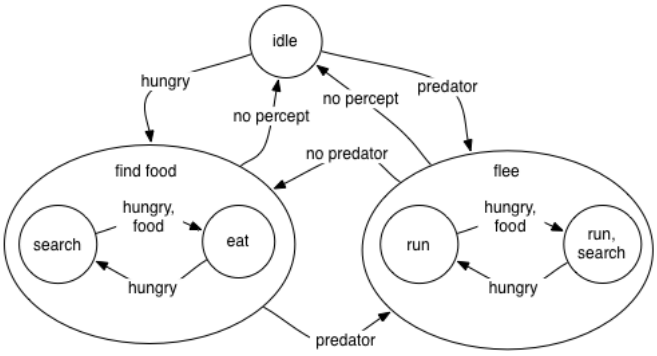

Fig. 3. Hierarchical finite state machine representation of the scavenger agent. Self transitions are not included for clarity. The total number of transitions in this case is 17.

Figure 3 shows the same scavenger agent, but presented as a HFSM. The HFSM reduces the overall complexity of the agent by creating three high level states instead of five. The *find food* and *flee* superstates are each composed of two state FSMs. While the total number of states increases, the overall number of transitions is reduced to 17, since the search, eat, run, and run/search states are no longer fully connected. The overall representation of the agent is simpler to understand, and it is easy to see that the super-states could be treated as separate modules to be inserted into other characters.

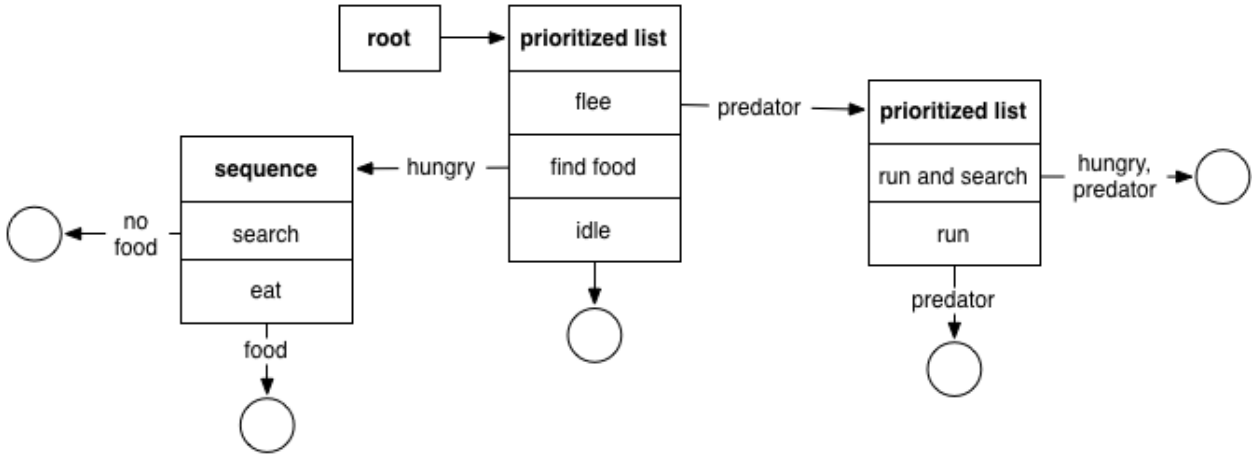

Fig. 4. HFSM-style behavior tree, composed of prioritized lists and sequences. The use of these structures reduces the number of transitions, but the complexity of the individual elements is greater. The number of transitions is 8, but this does not account for the full complexity of the model.

As a behavior tree, this agent is simpler still to understand, as seen in Figure 4. In this case, the agent is a tree with four internal nodes and five leaf nodes. The leaf nodes represent the primitive behavior to be executed, while the internal nodes represent decision points. The main decision to be made is first to choose from the prioritized list providing the high level behaviors of flee, find food, and idle. If a predator is seen, flee is chosen, and another prioritized list decision is made. If no predator is seen, it falls through to the find food branch, which will activate if the agent is hungry. If the find food branch is entered, the agent will simply activate search and eat in sequence. The priorities simplify the decision that is made at each stage, and eliminate the need for transitions between siblings. There are only seven explicit transitions in this case, though the individual nodes are more complex.

HTN-style behavior trees are very similar in structure to HFSM-style behavior trees, but may be executed differently. Depending on implementation, rather than performing conditional checks in the selector nodes, conditionals may be checked at the leaf nodes. Execution is then performed as a greedy search through the tree. While execution is different from the HFSM-style behavior tree, the structure of the tree itself is very similar, and so will be treated as the HFSM-style behavior trees for the sake of this analysis.

The subsumption agent in Figure 5 is even simpler. The available actions are listed in order of importance, and each is assigned an activating stimulus. The circles to the right of the layers represent the subsumption policy. The top layer cannot run simultaneously with eat or idle, so it must provide two override policies. The eat layer cannot run simultaneously with search or idle, so it also provides two. Finally, the search layer only specifies that it cannot run simultaneously with idle. This is a total of five suppression policies that must be specified, and three stimulus triggers (since the bottom layer must always be active if no other layer is active).

More complex subsumption architectures can be created using hierarchical subsumption, which allows a single top-level layer to be composed of two or more sub-layers. To demonstrate hierarchical subsumption, a more complex example is needed. For this example, the creature being controlled can see much further than it can reach, so if food is seen, it may be far away. To control this agent, when food is seen, first the food must be approached, then it can be eaten. This approach reduces the number of overrides that must be specified; if this was added as an additional layer, up to an additional four subsumption override policies would need to be added, but by making it hierarchical, only a single additional policy is required.

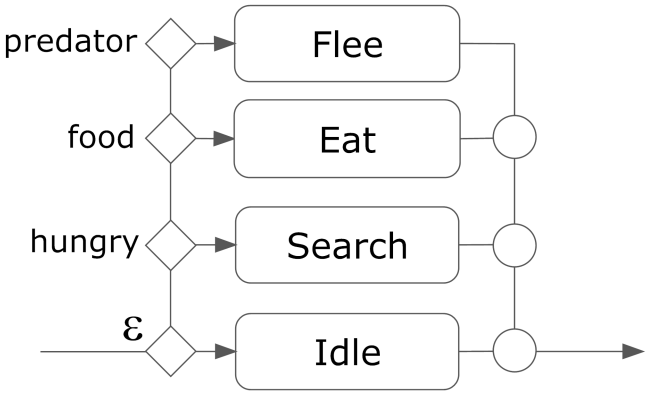

Fig. 5. Subsumption architecture representation of the scavenger. Counting triggering conditions and the individual components of the subsumption policies, there are 8 transitions. Three for the *predator*, *food*, and *hungry* triggers, two for the *flee* subsumption policy (since search can run concurrently), two for the *eat* subsumption policy, and one for the *search* subsumption policy.

A. Practical Implications

The representational complexity of different architectures can guide the choice of architecture for behavior editors. Finite state machines, with their very high complexity, are not appropriate choices unless agents are expected to be very simple, but in those cases provide an intuitive method

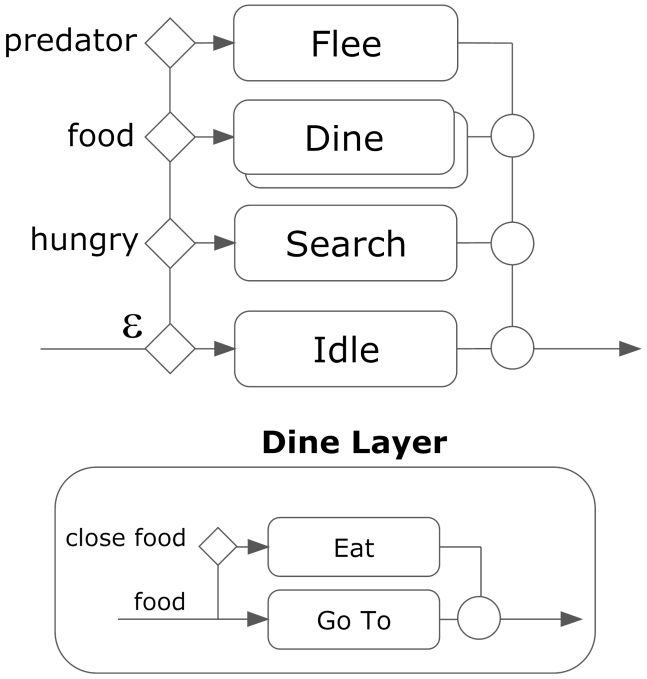

Fig. 6. Hierarchical subsumption architecture representation of the scavenger with limited reach. The two layers composing the *dine* hierarchical layer only add two transitions: one for the *close food* triggering condition, and one for the *eat* subsumption policy overriding *go to*. In this case, the *eat* subsumption policy could be omitted since go to will not generate any actions once the agent has approached the food.

due to the simplicity of the components. HFSMs greatly improve on this, and HFSM derived techniques such as behavior trees further improve the process. Behavior trees are a strong representation for experts, but have additional structural elements that may be difficult for novice users to master.

The low representational complexity of subsumption architecture is a major advantage over FSMs, and a strong competitor for HFSMs and behavior trees. Certain behavior patterns (such as sequences and loops if there is no method for creating complex behavior layers) are more difficult to generate with subsumption, but inherent concurrency eliminates the need for methods to perform locking or scheduling of behaviors. The process of scripting for tight control over an agent's execution of behaviors is much more difficult with subsumption. Tight control may be necessary when, for example, agents are used for storytelling and to advance the plot elements in a game. Despite this shortcoming, for *novice* users, subsumption provides an abstraction with a shallow learning curve and a minimally complex representation.

B. Theoretical Analysis

In order to make our intuitions about the representational complexity more precise, we introduce some notation. We assume that we have a set of percepts P and a set of actions A that the agent can perceive and perform respectively. It follows that the specification of a reactive agent is a mapping of the form $P*$ to $A*$ where $P*$ and $A*$ are power sets with cardinality $2^{|P|}$ and $2^{|A|}$ respectively. Note that, in this particular setting the agent is able to perceive one or more percepts and able to simultaneously perform one or more actions. This is quite typical in the case of agents in a virtual world of computer games and training simulations. A simple example of this is an agent that perceives a friend wants to exchange greetings (waves or says hello) while walking towards a goal.

Given this setting, our intention is to investigate the complexity of the agent specification. In order to introduce a metric for such representational complexity, we simply count the number of entities and relationships between the entities for a particular framework like finite state machines or subsumption architectures. So, in the case of finite state machines the number of entities would be the number of states and the number of relationships would be the number of state transitions. In the case of subsumption architectures, the number of entities would be equal to the number of layers while the number of relationships would be equal to the number of overrides amongst layers.

We now compare the representational complexity of finite state machines and subsumption architectures by comparing the number of entities and relationships in an agent specification. As far as the number of entities are concerned, it is easy to see that a finite state machine would require one state for every possible subset of actions. The number of entities would therefore be equal to $2^{|A|}$. Of course, in certain cases, it may not be possible to perform all actions simultaneously, so this is a worst case estimate. In contrast, observe that subsumption architectures would require a total of A layers as they inherently allow for parallel execution of actions. As far as relations among entities are concerned, as there are $|P|$ percepts and the agent can perceive any subset of them, a finite state machine will have a total of $|P| \times 2^{|A|}$ transitions. HFSMs may have an equally large number of states, and (in the worst case) may grow as large as a standard FSM if no abstraction of the behavior is possible. The best case for an HFSM may be better than that of its FSM equivalent, assuming that the specified behavior can be decomposed hierarchically, but the main improvement for the HFSM is in the number of transitions in the model.

In the case of hierarchical finite state machines the explosive increase in the number of transitions is controlled by grouping states into super states or meta states. The intuition is to group states that have relatively high intrastate transitions and relatively low interstate (at the meta or super level) transitions. By grouping states in this manner the number of transitions to be specified can be largely reduced. This reduction in the number of transitions is highly domain dependent as in certain problem domains it is possible to exploit the natural hierarchy among states, while in others there may not be any state reduction as it is not possible to group states. So in the worst case scenario there might be as many states as a finite state machine, but in the best case scenario it is possible to organize the underlying finite state machine as a structure that largely resembles a binary tree. In

such a case it is possible to group states such that each meta state has two child states, resulting in a drastic reduction in the total number of transitions.

To get a quantitative measure of the number of transitions in such a HFSM consider the following reasoning. First, note that there are three distinct types of transitions, parent-child transitions, sibling transitions, and self transitions. If the underlying finite state machine contains a total of n states and is organized as a binary tree, there are a total of $2(n-1)$ parent-child transitions. In addition to this, we have a total of $(n-1)$ sibling transitions and a total of $n-1$ self transitions. Thus the total of $4n-4$ number of transitions, which is a large reduction from the n^2 transitions in a FSM. It is important to note that this is a best case scenario and in practice the number of transitions in a HFSM is between $4n-4$ and n^2, depending on the problem domain.

Behavior trees have more complex elements than HFSMs, and the behavior tree technique differs from the others discussed because it is as much a set of common design and engineering principles as it is a control method. Though behavior trees may differ greatly across implementations, the main building blocks are still behaviors, conditionals, and parent/sibling relations. The parent/sibling relations are compounds instead of simple relations, and can be reduced. The *sequence* relation can be reduced to a parent relation and a set of direct sibling relations, so that for a parent with c children, there are c relations. *Selectors* are similar, and many of the decorator patterns such as *preconditions* and *parallels* have the same attributes. Therefore, while individual implementations will vary, the core behavior tree technique will have approximately $n-1$ relations for n entities in the best case balanced binary behavior tree. Note that a behavior tree with n entities may have fewer behavior or action entities; because conditionals are frequently stored as nodes in the tree, they are counted as entities rather than relations.

In the case of subsumption architecture, each layer will have a percept and action but the major component of the relationships among the layers is that of the override between the layers. Since each layer can override all layers below, the number of overrides is $(|A|-1)+(|A|-2)+(|A|-3)+...+(|A|-|A|)$. As there are a total of $|A|$ layers, there are $|A-1|$ triggering conditions (if the base trigger is not counted), so the total number of relationships (both triggering conditions and overrides) is given by Equation 1.

$$|A-1|+(|A|-1)+...+1=|A|^2-\frac{|A|(|A|-1)}{2}-1 \quad (1)$$

So a two layer subsumption agent has two relationships, a three layer subsumption has 5, a four layer subsumption has 9, and so on.

Consider the intelligent agent described earlier (see Figure 5). It has four percepts (including the null percept), and four predefined actions. Given these basic percepts and actions, the agent design has a number of key features that affect the representational complexity. First, note that each of the layers maps percepts to actions. For example, layer L3 maps the percept food onto the action eat. Furthermore, note that the layers are prioritized. If we assume that we do not want any layers to execute concurrently, Layer L1 is overridden by layer L2, which in turn is overridden by layer L3. Finally, layer L3 is subsumed by layer L4. Also note that layer L1 does not have a percept and corresponds to a default action that is performed when no percepts are received by the agent.

This simple agent designed using behavior-based control has the following overall behavior. When there are no percepts available, it maintains an idle state. When the L2 layer is triggered by hunger, the agent explores new regions. In the case where the agent perceives food, the L2 layer is overridden by the L3 layer, and the agent consumes the food. If the agent perceives a predator, the L4 layer is triggered, the L3 layer is overridden, and the agent flees from the predator. Note that the prioritization of the layers is the most important part of the agent definition and that higher layers can selectively override lower layers (combinations of layers producing blended actions are permissible). In addition, the L3 layer, in overriding the L2 layer, does not necessarily override the L1 layer.

While the example discussed here is only for the purposes of illustration, it does demonstrate the key aspects of behavior-based control, namely, the mapping of percepts to actions via layers, the prioritization of layers, and the ability to override or combine actions from multiple layers. Subsumption architectures have a number of advantages over the use of other architectures for game agents, such as finite state machines (FSMs), hierarchical finite state machines (HFSMs), and behavior trees. Behavior-based control is inherently parallel, as multiple active behaviors can be run at once. The representational complexity of a behavior-based control agent, which is an important consideration for the agent authoring process, is far lower than other architectures. As shown in Figure 2, the corresponding finite state machine for our hungry agent requires many more transitions. If multiple behaviors are allowed to be active at once, the finite state machine becomes increasingly more complex, as each allowable combination of behaviors requires an additional state. HFSMs and behavior trees reduce this complexity over simple FSMs, but still require more complex models to represent the same agent due to the increased number of states. The reduced complexity of behavior-based control makes it simpler and faster to create intelligent agents.

If *hierarchical* subsumption is compared to hierarchical finite state machines and behavior trees, an additional improvement is seen. The number of transitions for this case is better than HFSMs. To correspond to the best case FSM, consider a *binary subsumption*. In a binary subsumption architecture, every layer is a hierarchical layer composed of two layers (so that if the layers are depicted as a tree with parent/child relationships, it would become a binary tree).

In this ideal case of a binary subsumption, the only transitions are between siblings. Because one layer of each sibling pair is the base layer, it has no transitions. This leaves only

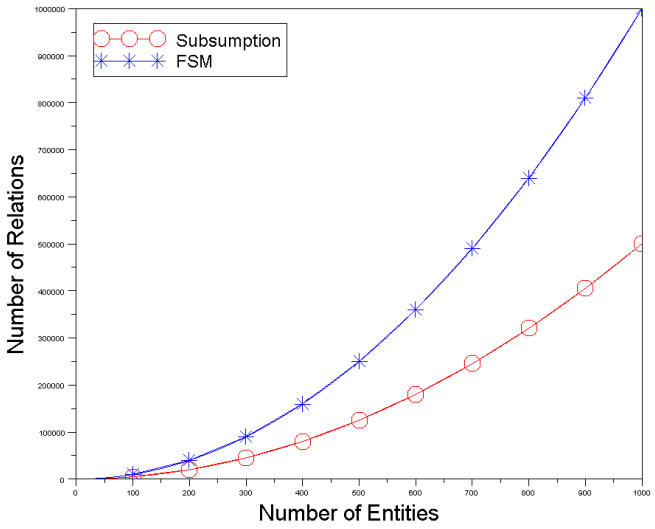

Fig. 7. Growth of number of relationships between entities as the number of entities increases (non-hierarchical models). Y scale is 0 to 1,000,000

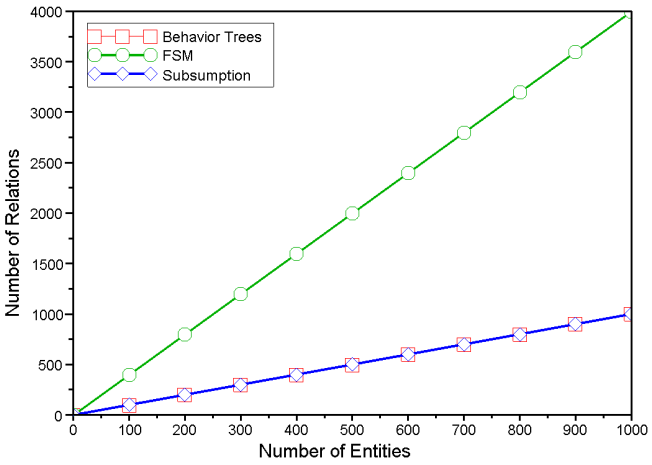

Fig. 8. Growth of number of relationships between entities as the number of entities increases (hierarchical models). Y scale is 0 to 4,000. The behavior tree and hierarchical subsumption growth is very similar, causing their graph to overlap.

one override relationship and one trigger condition per sibling pair, creating n transitions in hierarchical subsumption. Like HFSMs, the worst case—all layers at the top level—is identical to standard subsumption. It is also important to note that this best case is not a realistic scenario, and it is difficult to estimate an average subsumption controller, so the expected number of entities will generally be much higher.

The growth of the upper-bound worst case number of relationships (transitions or overrides) between entities with the number of entities is visualized in Figures 7. Figure 7 compares non-hierarchical finite state machines and subsumption architectures. For FSMs, the worst case is a fully connected controller, while for subsumption, the worst case is a controller in which each layer fully subsumes all layers below. Best case for each architecture is simply the number of entities. This is equivalent to a FSM that runs a series of actions and stops, or a subsumption controller that allows every layer to run at the same time.

In Figure 8, the hierarchical models are assumed to be binary trees, which provides the best case number of relationships for the hierarchical models. The hierarchical layers are assumed to be fully connected (two transitions for HFSMs, one override relationship for hierarchical subsumption). Hierarchical models that are poorly balanced have relationship growth similar to their non-hierarchical counterparts, except for behavior trees, which do not degrade strongly from their best case (though in practice, behavior tree complexity will grow in other ways). Note that the growth of entities as A increases will produce higher numbers of entities for HFSMs than hierarchical subsumption, even though the relationships for an ideal HFSM grow slower than the relationships in non-hierarchical subsumption.

C. Practical Application

Representational complexity should be viewed as a guide to help choose between reactive architectures. Each of the architectures considered in this paper have similar capabilities,but models are built very differently. Finite state machine architectures provide a very simple set of elements and paradigm for creating intelligence, but if large agents are required, the complexity of the model may grow too great. Behavior trees provide a great deal of flexibility and expressivity for advanced users who are trained in the use of its decorator patterns and other features, but are not as easily accessible to novice users.

Furthermore, representational complexity may affect models created in other ways; if models are learned, there is a trade-off between the simplicity of the input grammar of the model and the size. It may be more difficult to learn behavior trees than finite state machines because there is a larger set of possible options for each element of the model, but the final model will have fewer entities. This aspect of representation needs to be explored more fully to consider the complexity of the individual elements before any strong conclusions can be made.

Other factors may also affect the choice of architecture to be used. FSMs are simple to implement, and FSM/HFSM-based systems make it easier to create sequences of actions than subsumption-based systems. This allows tight control over agents in situations where scripts might otherwise be used, but more responsive behavior is desired. In some cases FSMs may have fewer relations than subsumption controllers, such as when the controller is primarily composed of state sequences that rarely branch, as may occur with scripted characters—in these cases, FSMs are a better choice than subsumption.

Subsumption provides inherent parallelism and lower overall complexity in the model, but there are still some concerns about managing emergent behavior. If the action model of agents being created is simple, emergent behavior is less of an issue, making subsumption a god choice for novice users. Behavior trees are commonly used standard throughout the

game industry, making it a widely understood architecture. If expert users are the target, behavior tree and HFSM editors can be quickly implemented using standard tree control widgets available in most graphical toolkit libraries. These editors will be very basic, but provide a substantial improvement for users who are well-versed in the architectures.

Many factors do contribute to the choice of architecture, but representational complexity can contribute a useful metric of the difficulty involved in managing agent definitions.

IV. Conclusions

In this paper, we introduced a concept of representational complexity for comparing different architectures for reactive agents. Representational complexity is a metric to assess the difficulty of authoring agents, and is a useful concept for evaluation of behavior authoring tools. As the demand for more life-like and interesting AI at larger scales increases, it will be important not just to develop authoring tools that work, but authoring tools that substantially decrease the authorial difficulty of agents. While game developers are the immediate target for behavior authoring tools, novice users should be considered as well. Behavior authoring has a strong role to play in future user-generated content as well as in the development of training simulations, and this will require tools accessible to non-experts.

Concepts such as representational complexity can help determine the best architecture for a particular tool; while behavior trees may be appropriate for experts, they may not work as well for novices. Ultimately these tools must be tested with real users, but choosing the right representation before development can save time and money.

V. Acknowledgments

This material is based on research sponsored by the US Defense Advanced Research Projects Agency (DARPA). The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the US Government.

References

[1] E. Bangeman, "Growth of gaming in 2007 far outpaces movies, music," http://arstechnica.com/gaming/news/2008/01/growth-of-gaming-in-2007-far%-outpaces-movies-music.ars, January 2008.

[2] R. W. Hill, J. Belanich, H. C. Lane, and M. Core, "Pedagogically structured game-based training: Development of the elect bilat simulation," in *In Proceedings of the 25th Army Science Conference*, 2006.

[3] R. Straatman, T. Verweij, and A. Champandard, "Killzone 2 multiplayer bots," Slides from http://files.aigamedev.com/coverage/GAIC09_Killzone2Bots_StraatmanChamp%andard.pdf, June 2009, retrieved May 4, 2010.

[4] Valve, "Left 4 Dead," PC Game, 2008.

[5] M. J. Matarić, "Behavior-based control: Main properties and implications," in *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, 1992, pp. 46–54.

[6] D. Isla, "Handling Complexity in the Halo 2 AI," in *Proceedings of the 2005 Game Developers Conference*, 2005.

[7] C. Hecker, "Liner Notes for Spore/Spore Behavior Tree," http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Doc%s, 2007.

[8] A. J. Champandard, "Behavior Trees for Next-Gen Game AI (video)," http://aigamedev.com/open/articles/behavior-trees-part1/, 2007, march 31, 2010.

[9] I. Millington and J. Funge, *Artificial Intelligence for Games*. Morgan Kaufman, 2009, ch. 5.4: Behavior Trees, pp. 334–371.

[10] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, Mar 1986.

[11] C. Cote, D. Letourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran, "Code reusability tools for programming mobile robots," *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 2, pp. 1820–1825 vol.2, Sept.– Oct. 2004.

[12] Artificial Technology, "Eki One," http://www.eikone.com, 2009.

[13] Xaitment, "xaitMove and xaitKnow," http://www.xaitment.com, 2009.

[14] Havok, "Havok Behavior," http://www.havok.com/index.php?page=havok-behavior, 2010, january 21,2010.

[15] Stottler Henke Associates, "Simbionic," http://www.simbionic.com/, 2004, retrieved November 23, 2008.

[16] Crytek, "Cryengine 3," http://mycryengine.com/, 2010, january 21, 2010.

[17] B. Schwab, "Implementation Walkthrough of a Homegrown "Abtract State Machine" Style System in a Commercial Sports Game," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008, pp. 145–148.

[18] Realmware Corporation, "Visual3d game engine," http://game-engine.visual3d.net/, 2010, january 21,2010.

[19] A. J. Champandard, "What Does a Behavior Tree Editor Look Like?" http://aigamedev.com/open/articles/behavior-tree-editor-example/, 2008, january 21,2010.

[20] F. W. P. Heckel, G. M. Youngblood, and D. H. Hale, "BehaviorShop: An Intuitive Interface for Interactive Character Design," in *Proceedings, 5th Artificial Intelligence for Interactive Digital Entertainment (AIIDE 2009)*, 2009.