Workflow via Avatarbots

Weston Barger, Kjartan Kennedy, Sarah Marsh, Grant Slatton, Taylor Yust

Abstract

Computers do not have a good way of representing higher level human activity. Thus, our objective was to represent workflows in a virtual world through the use of avatar bots. We accomplished this through a combination of behavior trees and a system composed of atomic workflow steps within the Unity video game engine. We also developed a grammar to map workflow representations and a system to parse them into trees. Finally, we explored the applications of Prolog in recognizing workflows.

1. Introduction

1.1 Problem

Computers currently do not have a good way of representing workflows in a computational manner that can be applied to any context, nor is there an intelligent means for virtual agents to select and execute appropriate workflows. In addition, computers lack the ability to recognize and categorize workflows in a way that would allow them to better understand underlying truths.

1.2 Objective

Our team’s objective is to represent workflows in a virtual world through the use of avatar bots within the Unity video game engine, and to be able to parse observed workflows into a meaningful format.

1.3 Context

Workflow representation has clear connections to AI. Intelligent agents often need to execute strings of actions that collectively contribute to some higher level goal. These workflows often employ the use of other AI techniques like behavior trees and pathfinding. Recognizing and parsing these workflows is also an application of AI.

This project is also tied to pervasive computing and 3D virtual worlds. In the “everything is alive” approach, we think of objects not as islands that exist independent of one another, but as interconnected and intelligent agents that can communicate and reason with one another. One way of representing this is through avatars in 3D virtual worlds like Unity. This representation requires some way of computationally representing and understanding workflows in order to execute.
2. Related Work

2.1 Key Technologies

**Unity** – Unity is a free video game engine. It can be used to construct 3D virtual worlds and supports the C#, JavaScript, and Boo programming languages. It is relatively easy to use and iterate with due to its wide support and play mode feature (which allows the simulation to be edited during runtime). The engine became our platform of choice for the development of our workflows.

By itself, Unity does have a few limitations, however. There is no built-in support for visual editing of complex webs, graphs, and relationships between objects such as the workflows in our simulations, so we had to rely on linear groups of steps that were somewhat unwieldy. There was no free built-in source control support, so we had to use Git as an outside source control tool. Prolog and Lisp, which are great for AI applications like this, are not natively supported in Unity. While Unity has support for streaming between “scenes” (or “levels,” i.e. groups or arrangements of game objects and their states), there is no standard means of streaming scenes during actual gameplay without developing our own methods. Built-in pathfinding via navigation meshes is also not supported in the free version, so we had to default to outside tools or more simplistic methods.

**Behavior Trees** – Behavior Trees are advanced tree structures that derive much of their basic functionality from finite state machines. The basic unit of behavior trees, instead of being a state, is an atomic task. These tasks are combined, using specific implementations of tasks including sequencers, selectors, and decorators, to build up complex behaviors for an agent. Because every element of a tree inherits from the same task structure, the interface will always be near identical, making it easy to visually build up complex trees for complex behaviors. Sequencers are a type of task that performs each of its child tasks in a set order, while selectors are another type that selects and performs a single child task.

For this project, behavior trees are used to wrap around Taylor Yust’s workflow representation, in a way so as to incorporate the usage of complex workflows in a more complex behavior for an agent. This would allow, for example, an agent to go about its day and when it comes to certain tasks it can execute the workflow to go with that task.

**“Everything is Alive”** – EiA is the idea that everything can be considered some sort of “smart” object that is aware of its state and function as well as its relationship to the rest of the world and other smart objects. Instead of serving simple mechanical functions, devices can be “communicated” with to perform certain functions, or they might “reason” about the world and perform functions automatically. This “communication” and “reasoning” requires the implementation of artificial intelligence to drive these behaviors.

The representation of workflows extends from the EiA concept. Smart objects can be represented in a virtual world as executing higher level workflows in an intelligent manner. These workflows need to be represented in a way that can be computationally understood.

**Recursive Descent Parser (and Generator)** – A recursive descent parser is a relatively simple and straightforward method for parsing context free grammars. It is relatively simple to construct a recursive descent parser from a Backus-Naur Normal Form Grammar (BNF Grammar).
Genetic Algorithms – A genetic algorithm is a stochastic optimization technique that is inspired by natural evolution. By representing a solution as a genome, it is possible to breed and mutate these genomes to produce new, potentially better solutions. By iterating from generation to generation, the average fitness of the population tends to increase.

2.2 Related Work

Step System - Taylor Yust, while working on Dr. David Fredrick’s Mythos Undbound project, was tasked with designing and developing an event system, which became the precursor to the workflow representation for this project. Yust’s step system used many of the same elements as the workflow representation and provided a solid foundation to build upon. The step system is currently used in the game lab to build up complex events in order to advance story or set up gameplay elements.

Information about implementing game AI techniques in Unity, including state machines and behavior trees, came from Unity 4.x Game Programming. [1]

2.3 Related Class Projects

Our project on workflow representation relates to these other class projects:

- Workflow Logging, Querying, and Inventory – Our workflows needed to output to XML formats in a way that this team could interpret and analyze, which we were able to accomplish.

- Virtual Representation of Smart Objects – Our workflows needed to be able to interface with arbitrary smart objects in a generic way, which we accomplished through the implementation of our WorkflowSteps.

- Architectural Representation for Virtual World Buildings – Avatar bots exist in a virtual space and thus need a virtual environment to navigate. This team constructed the virtual worlds we used for our workflows, and we ensured that our bots were capable of navigating them.

- Workflow Visualization – Workflows cannot be constructed or visualized without data structures or objects representing the underlying workflows. Our project ensured that the visualization team can build workflows using our tools.

3. Architecture

3.1 Requirements

- Software must be able to represent workflows in a generalized way that can be applied to any context.

- Higher level workflows should be able to be constructed from lower level workflows.

- The selection and execution of workflows by agents should be intelligent.

- Workflows should be able to interpret and navigate virtual environments.

- Workflows should be able to interact with smart objects in a virtual world.
• Workflows must output a log of meaningful data.
• Workflow logs should be able to be parsed and better understood by computers through critical analysis.
• Workflows should be able to be represented in an XML format.
• Workflows should be capable of being represented in defined application-specific XML formats, such as a recipe.
• Workflows should be able to be specified by a context free grammar.
• Behavior Trees should be easy to construct from basic tasks.
• Behavior Trees should select and execute tasks logically and effectively.
• Behavior Trees should be able to execute workflows.

3.2 Architecture

Workflow Representation – Workflows are represented as collections of atomic units that are each derived from WorkflowStep. Each WorkflowStep represents some sort of small action in a larger workflow, such as rotating towards a destination or picking up an object. Each step has StartStep(), UpdateStep(), and FinishStep() methods that control initialization, execution, and finalization, respectively. StartStep() initializes all the variables the agent will need to execute the action, such as finding references or calculating destination values. UpdateStep() is called every frame, returning true when the agent has accomplished the step’s objective and false otherwise. FinishStep() wraps up any last calculations and sets all values to their final states before calling the next WorkflowStep in the workflow. WorkflowSteps include inputs and outputs that feed into one another in order to carry data around larger workflows.

WorkflowBase, derived from a base Workflow class, encompasses the lowest-level workflows and are made of only WorkflowSteps. WorkflowBase manages the execution of its WorkflowSteps and ensures they accomplish their objectives in a linear order. WorkflowGroup is also derived from Workflow, but, unlike WorkflowBase, is composed of only other Workflow objects. It controls the execution of the workflows it manages, executing them in a linear order similar to how WorkflowSteps will execute in order. In this way, lower-level workflows can be built up from WorkflowSteps and WorkflowBases while WorkflowGroups can represent ever higher levels of workflows as they are built on top of one another.

WorkflowSteps, using the Unity engine and its various tools, can interact with other objects and their defined methods. For example, a WorkflowStep could tell a given Blender object to Blend(), which will cause the contents in its BlenderContents[] array to blend and merge into a single resulting BlendResults object. Unity can also be used to calculate positions and vectors in a 3D space to allow agents to move around the environment at given speeds every update frame.

Behavior Trees - The behavior tree system has two primary components: the Mind, and the Task. The Mind is attached to an agent, and is given a root task for the behavior tree to execute. The Task is an abstract class, and represents an atomic task. It consists of StartTask(), Run(), and EndTask() methods that allow setup, execution, and closing down of a task. As a task is executing, it returns one of three codes on each update cycle, in order to alert the mind as to what to do on the next cycle: Succeeded, Failed, or Running. These are fairly intuitive to understand based
on the description given for workflow representation. Within the behavior tree system is an interface between itself and workflows, which takes a Workflow object and executes it. This allows the task to return a code similar to what the Workflow object is returning, in order to allow the Mind to continue executing the Workflow without continuing on to other tasks.

**Workflow Logs** – A WorkflowLog object can be placed into any scene that wishes to log its output. By default, the Workflow and WorkflowStep classes include virtual methods that search for a WorkflowLog object and will output execution with timestamps to it in the form of strings. Default messages are provided, but derived WorkflowSteps can override this method to include their own execution messages.

**Workflow XML** – Using a custom editor script, a user can click a button on any workflow object to generate an XML representation of it. The code recursively searches the workflow’s component Workflow and WorkflowStep objects, including all inputs and outputs, and arranges them into a hierarchical XML.

The workflows were also designed to be reinterpreted for any domain-specific application. Different contexts are going to require their own specific XML formats that aren’t covered by the default representation of workflows. Interpreter scripts can be written to comb through workflows and reformat their data for their own purposes. As an example, our final project includes a recipe interpreter that searches for inputs with “Ingredient” or “Equipment” tags and organizes recipes with lists of requirements and recipe steps, all represented in XML in a structure that could be understood by the XML team.

**Prolog** – A kitchen workflow was represented using Prolog. A recipe was defined in a declarative way so that it could then be queried. Using Prolog a recipe was defined as follows:

\[\text{recipe}(\{X,Y\}) :\text{- list}\_\text{ingredients}(X), \text{list}\_\text{prepare}(Y).\]

i.e. a recipe is a list with two elements, both themselves lists. The first element is an ingredients list and the second is a food preparation workflow. The food preparation workflow is a list containing actions. An action is itself a list with three elements:

\[\text{action}(\{X,Y,Z\}) :\text{- verb}(X), \text{noun}\_\text{ingredient}(Y), \text{noun}(Z).\]

The actions are made more specific with further definitions and classified by type of action. For instance, and action could be a “baking action”. The definition would look like,

\[\text{action}(L) :\text{- action}\_\text{baking}(L).\]

\[\text{action}\_\text{baking}(\{X,Y,Z\}) :\text{- verb}\_\text{baking}(X), \text{noun}\_\text{ingredient}(Y), \text{noun}\_\text{appliance}\_\text{baking}(Z).\]

where we might have

\[\text{verb}\_\text{baking}(\text{bake}).\]

\[\text{noun}\_\text{ingredient}(	ext{chicken}).\]

\[\text{noun}\_\text{appliance}\_\text{baking}(	ext{oven}).\]

In this case, the queries

\[?- \text{action}(\{\text{bake, chicken, oven}\}).\]

\[?- \text{action}\_\text{baking}(\{\text{bake, chicken, oven}\}).\]
both yield true. The following is an example of a recipe (although it is not what the output of the implementation would yield),

?- recipe(L).

\[
L = \text{[[chicken,tomato],[[cut,chicken,knife], [cut,tomato,knife], [bake,chicken,oven]]].}
\]

In the actual implementation, there were several more constraints placed on a recipe. For instance, it needs to be the case that all of the ingredients used in the the workflow list were included in the ingredients list. Furthermore, we want to eliminate actions like

\[
\text{[cut,chicken,spoon].}
\]

So, a can_use(X,Y) function was added to explicitly state which tools can be used in which ways. The second version of the implementation includes a way to write the recipes found to a text file.

**Workflow Parsing and Generation** – Workflows are represented as a context free grammar in Backus-Naur Normal Form, such as:

cookies = gather_ingredients make_dough prepare_tray begin_baking wait prepare

gather_ingredients = buy_ingredients | "get_ingredients_from_cupboard"

buy_ingredients = drive "purchase_ingredients" drive

drive = "drive" drive | "drive"

make_dough = make_cream finish_dough

make_cream = "sugar" "butter" "shortening" beat

beat = "beat" beat | "beat"

finish_dough = "baking_soda" "eggs" "vanilla" "flour" beat "chips"

prepare_tray = "pam_spray" ball_dough

ball_dough = "ball_dough" ball_dough | "ball_dough"

begin_baking = "set_oven" "insert_into_oven" "set_timer"

wait = "wait" wait | "wait"

prepare = "remove_from_oven" "unset_oven" put_on_plate

put_on_plate = "put_on_plate" put_on_plate | "put_on_plate"

Where quoted items are atomic and unquoted actions represent production variables. A grammar is composed of multiple lines consisting of a left side and a right side. The right side consists of multiple expressions, which are then made of variables or atomic strings. Thus, the grammar can be represented as an array of 2-tuples, in which the second element is an array of arrays of tokens (in type notation: [[variable, [[variable|atomic]]]]). Alternatively, our system also accepts inputs in the plaintext format shown above.

The grammar generation using genetic algorithms represents the grammars in the array format previously described. Possible mutations are: Insert random line, delete random line, duplicate line, insert random expression, delete random expression, duplicate expression, insert random token, delete random token, and duplicate token. These mutations, combined with genome crossover, allow any grammar to be generated in theory.
Grammar discovery works by using the parser generator to generate parsers for genetically created grammars, scoring them based on the parser’s performance, and culling the poorly performing genomes.

3.3. Tasks
Weston Barger
- Create grammar of cooking activities using Prolog.
- Use Prolog to recognize unique workflows.
- Use Prolog to generate new recipes from ingredients lists.
- Compose high level workflows from lower level ones.

Taylor Yust
- Create a data structure for workflow data.
- Create a representation of a workflow step.
- Create a workflow data structure that can be composed of workflow steps and other workflows.
- Export workflow logs and XML representations of workflows.

Kjartan Kennedy
- Create a wrapper for Taylor’s workflow representation.
- Create a manager for behavior trees used to execute workflows.
- Find a way to export a workflow to test workflow recognition.

Sarah Marsh / Grant Slatton
- Develop a grammar that maps to workflow representation.
- Develop a system to parse a string of leaf nodes into a full tree.

3.4 Testing
Testing for workflow representation and the behavior tree system was iterative. Due to Unity’s robust play mode and debug features, we were able to analyze simulations states in real-time and look for errors or discrepancies in data. This was useful in determining when certain Workflow Steps were not hooked up correctly, when workflows were sending the wrong outputs or receiving incorrect inputs, or when avatars were not navigating the virtual space correctly or according to their workflows, along with how tasks were executing and making sure that they executed correctly.

Logging and XML outputs were written to text files. These were compared to desired outputs (such as the XML team’s recipe XML format) and were used to adjust the code accordingly.

Prolog – The Prolog programs were tested incrementally. Each time a definition was added, the program was run and the output was analyzed to determine the effectiveness and the efficiency
of the addition. In many cases, things needed to be restated or “cut” so that Prolog could unify results in reasonable time. Many initial definitions caused full stack errors.

4. Results and Analysis

Workflow Representation – We were able to accomplish our objective of representing complex workflows in a hierarchical way that computers can understand and analyze. Given an arrangement of WorkflowSteps and higher level Workflows, an avatar bot can intelligently select a workflow from a behavior tree and execute it. For example, in the above screenshot an avatar bot can be seen moving to cut up a chicken with a knife as part of a larger cooking recipe workflow.

Behavior Trees - The structure for a behavior tree system was successfully implemented. As of now, we run into the same problem as we do with workflow, in which we must use an unwieldy hierarchy of empty Unity GameObjects to represent our behaviors. We can use the behavior tree system on its own to build up a complex behavior, or we can combine it with the wrapper for the workflow representation to integrate workflows into our behaviors.
The above-left screenshot shows how this workflow is represented within Unity using collections and hierarchies of game objects designated as workflows and steps. The above-right screenshot shows the XML representation of this same workflow.

As seen above, workflows are also able to correctly output logs that can later be parsed and analyzed. Default messages are built into every workflow and step, but they can be overridden and customized to a particular step’s needs.
The above screenshot is an XML representation of a recipe formatted as required by the XML team. This is an example of just one type of interpreter, but any arbitrary interpreter could be written for any number of applications.
Workflow Parsing – We successfully created a system that can take an arbitrary workflow as input and learn a compact grammar that will parse that workflow. Since genetic algorithms are relatively simple stochastic algorithms, the algorithm in rather unwieldy for large inputs. It does work for simple workflows, so as a proof of concept, the project was a success. As an example, consider the language of a vegetable preparation workflow. An example workflow might be:

```
get_vegetables chop chop chop chop chop chop chop chop chop
```

take the language of a vegetable preparation workflow. An example workflow might be:

```
get_vegetables chop chop chop chop chop chop chop chop chop
```

A grammar describing this workflow should learn that the “chop” action can be done an arbitrary number of times, and is surrounded by “get_vegetables” and “eat”. Initially the genetic algorithm produces a random grammar which fails to correctly parse the inputs:

```
False 0
( _0, [['_4', '_2'], ['_0'], ["eat", "get_vegetables", '3'], ["eat", "get_vegetables", '2'], ["_0"]])

( _1, [['get_vegetables'''], ['chop''', 'get_vegetables'''], ['eat''', 'get_vegetables'''], ['get_vegetables''', '_1', '_4']])

( _2, [['_0'], ['_4', 'chop'''], ['_2', 'chop'''], ['_1', 'get_vegetables'', 'eat''']]

( _3, [['chop'', '_1'], ['get_vegetables'', '_2'], ['chop'', 'eat'''], ['eat'', '_4'], ['eat'', '_2', 'eat''', '_4']]

( _4, [['chop'', '_0'], ['chop'', 'get_vegetables'', 'eat'''], ['_3'], ['_4', 'chop'''], ['get_vegetables'', '_1', '_2']])
```

But very quickly the genetic algorithm discovers the optimal solution:

```
True 0
( _0, ["get_vegetables'', '_1'])

( _1, ['chop'', '_1', ['eat'']])
```
5. Conclusions

5.1 Summary

The project was largely a success. We were able to represent workflows, execute them intelligently through behavior trees, and parse outputs to analyze workflows.

However, despite our successes, it became apparent that this project is just one small part of a much larger endeavor that must be taken in order to represent workflows in a universal and robust manner. Our project works for the cooking and recipe domain, but it needs to be expanded upon and applied to other domains.

In addition, while many teams were involved in the workflow project, in the end it was difficult to maintain full compatibility with one another, and as a result there is much work that has yet to be “plugged into” the rest of the project.

5.2 Potential Impact

While only just the start, the implications of a generalized means of representing workflows is great. It would contribute largely to the Everything is Alive movement by giving smart objects a common means of understanding and executing workflows. Applications might include modeling hospitals or health services to improve efficiency, or creating more compelling and intelligent AI agents in video games.

Prolog – The Prolog programs written were only an example (a kitchen) of a workflow representation, but they have meaning in a more general sense. In this case, the Prolog programs stated the definition of a workflow and then used search techniques to generate possible workflows. The more the definitions were refined, the more reasonable the possible workflows became. The usefulness of a representation is context dependent. Therefore, representing workflows as a set of rules governing possible actions is helpful for understanding how to represent a workflow.

5.3 Future Work

Workflow representation is done through Unity’s GameObjects for easier and quicker construction of workflows, but it would be more ideal to represent everything through XML instead of simply outputting to it. This sort of representation would allow better tools to be developed to create workflows, such as graphical web-like structures that better reflect the organization of workflows. This would also avoid the current implementation of workflow inputs and outputs, which is a workaround and very messy.

Behavior tree representation is handled similarly to workflow representation, and would be best handled if it was represented as XML as well. Just like for workflows, this would allow for setup and debugging of behavior trees to be significantly more intuitive.

We need a better, more standardized way of representing “smart objects” in order to better analyze workflows. We’re currently using tags that are applied outside of the objects, but the objects themselves should be queried to report “what” they are so workflows can make better use of them.

It is possible in Prolog to write a definition which using pure logic would yield the correct result, could take too long using Prolog’s search techniques. The definitions could be made much more efficient. Additionally, it would be nice to refine the definition further so that the recipes are
more realistic. Once that is done, the output could be put in a database of recipes so that another team could analyze it. Perhaps a team could use statistical analysis to determine probably next steps in a workflow.

It is clear that representing workflows as a context free grammar is useful in terms of parsing and discovering workflows. However, our method of using genetic algorithms to learn workflows from a corpus is highly inefficient, particularly for large grammars. Future work on this project should focus on learning grammars using a more deterministic approach.

5.4 Project Value

This project was selected by the team because it was at the intersection of a number of the team members’ interests. It utilized game AI techniques like behavior trees and workflows which could be applied to gaming agents, both of which are of interest to Taylor and Kjartan.
Bios

- **Taylor Yust** – Yust is a second year graduate student in the University of Arkansas computer science program. He has experience developing games for smartphone platforms and the Unity game engine. His involvement with Dr. Fredrick’s *Mythos Unbound* project led to the creation of the step system that served as the basis for workflow representation. Having attended multiple conferences (including the Game Developers Conference) with his gaming software, his background has made him uniquely qualified for this project.

- **Kjartan Kennedy** – Kennedy is a senior in the Computer Science and Computer Engineering department at the University of Arkansas. He works alongside Yust on the *Mythos Unbound* project as a designer and programmer. As a student of in classes including artificial intelligence, game design, and human-computer interaction, he has gained valuable insight to work on intelligent behavior in the Unity game engine. His capstone project will build upon several techniques that were prototyped in this project in order to build up a more robust and user friendly system.

- **Weston Barger** – Weston Barger is a senior candidate for a B.A. in Computer Science and a B.S. in mathematics. His background in mathematical proof and logic gave him unique insight into using Prolog to represent workflows.

- **Grant Slatton** – Slatton is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed algorithms and formal languages courses. This student will be responsible for implementing the parser generator and search algorithm. The documentation and testing will be shared among the team.

- **Sarah Marsh** – Marsh is a junior Computer Engineering major in the Computer Science and Computer Engineering Department at the University of Arkansas. She has completed an algorithms course. This student will be responsible for implementing the genetic algorithm and grammar checker. The documentation and testing will be shared among the team.

- **Dr. Craig Thompson, Mentor** – Thompson is a professor in the Computer Science and Computer Engineering Department. He leads the *Everything is Alive* research project that is currently focusing on how to simulate pervasive computing using 3D virtual worlds. See [http://vw.ddns.uark.edu](http://vw.ddns.uark.edu).

References

Appendix A – Deliverables Manifest

- Zip file containing Unity project file called WorkflowsFinal.zip.
- All relevant code for workflow representation can be found in the “Scripts” folder of the Unity project.
- All relevant code for the behavior tree system can also be found in the “Scripts” folder of the Unity project.
- Unity project also includes a scene called FinalDemo that can be found in the Scenes directory. Hitting the space bar while playing the scene will give a demo of a workflow. The workflow can be examined in the scene hierarchy.
- Unity project includes a scene called BehaviorTreeDemo which shows the structure and execution of a simple behavior tree.
- A folder of Python programs containing the code for the parser generator and genetic algorithm.

The unzipped Unity project file can be opened using Unity, which can be downloaded at the following URL: [http://unity3d.com/unity/download](http://unity3d.com/unity/download)

Source code can be examined within Unity itself, or it can be found within the Assets folder of the Unity project file.

Appendix B – Individual Team Member Contributions

Taylor Yust

- Taylor developed all of the workflow representation. He created the data structures that represented the workflow steps and their higher level workflows.
- Taylor ensured that higher level workflows could be composed of lower-level workflows.
- Taylor created the logger that tracks workflow execution and logs it to a text file.
- Taylor developed the XML representation of workflows and created a means for a workflow’s Unity representation to be converted into the XML format.
- Taylor developed the interpreter used for converting workflows into recipes that could be read by the XML team, giving an example of how such interpreters could be built.
- Taylor created a final demo scene tying together all of the above elements.
- Taylor provided support to the other teams for various questions related to Unity, workflow representation, and game programming.
- Taylor helped the Workflow Visualization team to become more familiarized with Unity and better understand how to visualize workflows through the step system.
Taylor provided resources and information to the Architectural Representation of Virtual World Buildings team on pathfinding and how to implement it in their environments.

Taylor worked with the Virtual Representation of Smart Objects team to help them better organize their project and structure their smart objects in such a way that workflows could better interact with them.

Kjartan Kennedy

- Kjartan developed the base system upon which to build the behavior tree system.
- Kjartan developed specific tasks that could be executed by the behavior tree mind.
- Kjartan developed a demo scene showing the structure and execution of a simple behavior tree.
- Kjartan developed a wrapper for Taylor Yust’s workflow representation to allow it to be executed as part of a behavior tree.
- Kjartan provided support for other teams with Unity questions.
- Kjartan communicated with the Smart Objects and the Virtual World Representation teams in order to make sure we could plug their deliverables into our demonstration scene.

Grant Slatton

- Grant developed the kitchen grammar specification.
- Grant developed the parser generator to create parsers from generated kitchen grammars.
- Grant made the parser generator API accessible to Sarah's genetic algorithm.
- Grant made the parser generators also display parse trees for analysis.

Sarah Marsh

- Sarah developed the genetic algorithm.
- Sarah determined possible mutations and grammar representation.
- Sarah devised a grammar scoring heuristic.
- Sarah integrated the genetic algorithm with Grant's parser generator to test and breed better populations of grammars.