Embedding Causal Block Diagrams Within Behaviour Trees

Bentley James Oakes*

McGill University

Abstract

Causal block diagrams are a formalism to model systems using mathematical values. However, their text output can be difficult to visualize. This paper aims to describe a system where causal block diagrams can be visualized using a Java simulation. This is achieved by embedding causal blocks within a behaviour tree formalism. With this new hybrid formalism, a number of advantages are realized such as increased flexibility and expressibility. Two simulations using these hybrid trees are presented, as well as a discussion of the new formalism's suitability to these problems. These simulation experiments are the circle test and a personal space simulation, and the resulting hybrid trees are also examined.

Keywords:
causal blocks, hybrid formalism, behavior trees, behaviour trees, modelling, simulation, Java, behaviour graph

1. Introduction

Modelling of a system may involve a certain amount of trial-and-error to fine-tune the model. With an appropriate way of visualizing the output, this process of validation may become much easier. By examining different hybrid formalisms, it may be possible to find a natural way of embedding one formalism into another in such a way to benefit from the structure of both formalisms.

*bentley.oakes@mail.mcgill.ca

This paper will examine the issue of modifying an appropriate formalism by embedding causal block diagrams. The formalism selected is behaviour trees, which offer a number of advantages described below. This embedding procedure will be accomplished by creating behaviour tree nodes that implement the same semantics as causal blocks. This hybrid formalism will then be visualized in a simulation, in order to provide immediate feedback to the model creator as the simulation evolves over time.

In section 2, related work will be described. Following this, section 3 will give an overview of the formalisms to be combined, while section 4 will explain how this embedding is achieved. A description of the experiments performed with results will follow in section 5. A comparison will be made with the work of others in section 6. Finally, a conclusion with future work will follow in section 7.

2. Related Work

Causal block diagrams have found use in describing a number of physical systems, as described in Denckla and Mosterman (2005), along with a formal definition of the formalism. This gives formal precision for the scenario where the causal block formalism is to be transformed or embedded into another. A similar definition for behavior trees is expressed in Colvin and Hayes (2007). A brief discussion of causal block diagrams can be found in Posse, de Lara, and Vangheluwe (2002), as well as a description of the visual modelling tool Atom3. This tool can be used to quickly build causal block diagrams, as well as many other formalisms. AtoM3 also provides some support for using a hybrid of formalisms inside one model.

Behaviour trees offer a compact and efficient representation of the possible actions for an agent in a simulation. Their flexibility has been demonstrated in a variety of contexts. Isla (2005) explains the use of behaviour trees to control agents in the combat-oriented Halo 2 by Bungie Studios. Each agent had their own behaviour tree which transformed local and global information such as player and squad position into a specific action for an agent. These behaviour trees were then customized for each one of a number of alien types within the game to differentiate their behaviour between 'cowardly' and 'aggressive' in combat. Hecker (2005) describes some modifications of the Halo 2 system. This new behaviour tree model was used in Spore by Electronic Arts. Agents in this game were not solely combat-oriented, but also had social behaviours as well.

Another interesting area of research is the application of genetic operations to behaviour trees, as explored in Lim (2009). Genetic operations are similar to the ones performed by DNA. In that genetic process, behaviour trees were copied repeatedly to form a population. Some behaviour trees were then randomly mutated to induce a shift in 'fitness' according to some metric, which was the tree's performance as a strategy in Introversion's DEFCON. In the real-world, Liu and Baltes (2004) describe a robotic soccer team architecture using behaviour trees. The flexibility and simplicity of behaviour trees is cited as an important factor in being able to change a robot's behaviour in a predictable manner.

3. Formalisms Used

3.1. Causal Block Diagrams

To model a system with mathematical relationships, it is natural to use causal block diagrams as seen in figure 3.1. For an explanation of causal blocks, Vangheluwe (2012) gives a detailed presentation. As a brief overview, causal blocks are nodes in a graph. At each time-slice, blocks produce some output as a function of their inputs. Computation flows from the leaf nodes, which do not rely on their child's values, to connected blocks further up the tree. This propagation of values is handled automatically by the causal block simulator, by determining which block's values need to be computed before others, and creating a dependency graph.

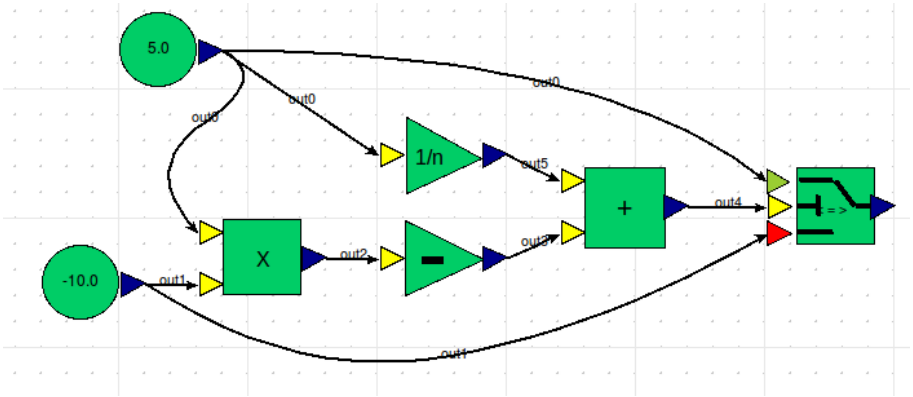

Figure 1: A sample causal block diagram

Denckla and Mosterman (2005) offers a formal notation for causal block diagrams, which shall be presented here in a simplified form. Let a causal

block be B, a four-tuple of $\{X, Y, F, E\}$. X is the set of inputs at a time-slice for a block. The block will execute the function $F(X)$ to produce $Y \in \mathbb{R}$. Then, for each edge $e \in E$, e transforms $Y \Rightarrow X_i$ for some other block B_i. These edges E are the set of edges to other blocks and are the connections that link together the causal block diagram.

$$B = \{X, Y, F, E\} \tag{1}$$

The mathematical blocks in a causal block diagram are typically very simple. Such blocks could include a summation block, where $F() = \sum$ and therefore $Y = \sum(X)$. Other blocks can also easily be expressed in this way. The negator block takes a single value and outputs the negative value of the input. In formal notation $F() = -X$ given that $|X| == 1$. Note that there is now a restriction on the cardinality of X. Another block of interest is the test block, which will output one of two values depending on an input value. This is done through the use of ports. Let X_{input} be the value defined to be connected to an input port. Similarily, X_{true} and X_{false} are values connected to the two other inputs ports in this block. This implies the X may need to be a mapping of port names to values. Then, at a time-step, $Y = (X_{input} > 0)?X_{true} : X_{false}$. Even though this block is powerful, note that the values for both X_{true} and X_{false} have to be evaluated at every time-slice. This is inefficient, and may lead to wasted computation.

3.2. Behaviour Trees

The notion of computation flowing up and down the tree in causal block diagrams can also be found in behaviour trees. In this formalism, nodes compute some function on their children, and output a value on an outward edge. However, in contrast to causal block diagrams, all values produced in the behaviour tree are booleans. As noted in the related work, this network of connections leads to a modular and flexible formalism. Note that functions in a behaviour tree typically rely on global side-effects. This is done to examine or affect the state of the simulation, such as looking for enemies nearby or playing an animation.

A formal definition of behaviour trees can be found in Colvin and Hayes (2007). However, most of these details are beyond the limited scope of this paper, though section 7 touches upon related future work. For this paper, it shall suffice to offer a simplified formal definition of a behaviour tree.

Behaviours trees are connected nodes, in some tree or graph form. Note that graphs are permitted, though cycles are not (without special conditions).

A behaviour tree node N is a four-tuple $\{X, Y, F, I, O\}$. Note that this a very similar definition to a causal block from above. However, some of the terminology does change. Edges in I are from a node to the children of that node. The children of N are defined to be the nodes N_i at the other end of each edge in I. X is the set of values coming from the children of N at a particular time slice. These values are received after the node N queries the edge in I which leads to that child. This child performs their own calculation and returns a value to N.

As mentioned before, values in a behaviour tree are booleans, implying that each $x \in X$ is also $\in \{0, 1\}$. Similarily, $Y \in \{0, 1\}$. The function F can be any arbitrary function, but it often has global side-effects on the simulation. This is the greatest dissimilarity from causal blocks diagrams, and will be discussed below. The final piece of the tuple is the twin set of edges I and O. This is to enable the node to query its children for their values. The edges in the set O are from a node to its parent or parents. These edges transform the node's output Y into X_i for some parent node i. Note that in this paper's implementation, a node cannot be the parent/child of itself. This is an example of a cycle as described above.

$$N = \{X, Y, F, I, O\} \tag{2}$$

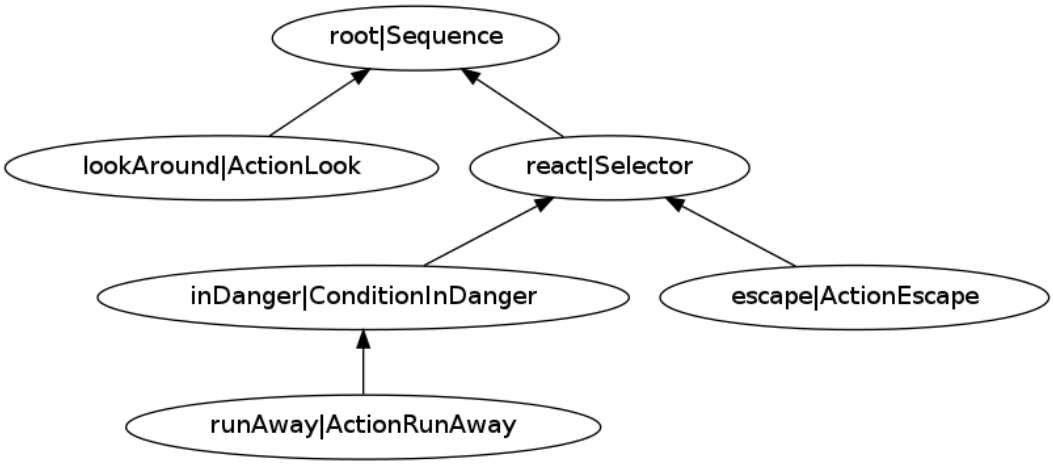

Figure 2: A sample behaviour tree diagram

Each node in behaviour tree can be quite simple or arbitrarily complex, with many side-effects. Four nodes will be briefly presented here, and exam-

ples may be seen in figure 2. The first is the sequence node, which has some particular order to its children. It queries each child in I in a particular sequence. If at any time the x received from a child is false, the sequence node returns false. If all children return true, the sequence node returns node. This is in contrast to the selector node, which queries its children in turn, and returns true if any child returns true. The selector node only returns false if all child nodes have returned false. It is also worth noting that there exist randomized versions of the sequence and selector nodes, which query their children in a randomized order.

Condition nodes, a third type of node, simply run some test against the state of the world. They return false if the test failed. If the test succeeded, the condition node returns the value of its child node. In this way, the condition node acts as a gate for control flow. This could have large performance considerations, as an entire subtree could be not evaluated. The last type of node is the action node. This node performs an arbitrary function such as movement or animation, and returns whether or not the action was successful. This node provides a large degree of power to the behaviour tree designer, as new action nodes can be created easily and placed into a tree.

Further examples and explanations of behaviour trees can be found in Lim (2009).

3.3. Formal Transformation

It is beyond the scope of this paper to define a formal translation method between a causal block diagram and a behaviour tree. This is further discussed in the future work subsection of this paper. However, it is informative to briefly examine the transformation from one causal block to a behaviour tree node, using the formal syntax above. This transformation is what allows the embedding of causal blocks in behaviour trees, leading to the hybrid tree advantages discussed elsewhere in this paper. The following section also describes the specific causal blocks that have been implemented in this paper. This embedding procedure has been conducted in an non-rigorous way, but has been informed by the details in this subsection.

3.3.1. Block to Node Mapping

For the transformation between a causal block and a behaviour tree node, note that 1 and 2 are very similar constructs. Indeed, the input set X, output set Y and function F are a one-to-one mapping. It is only in the connections to other blocks that some processing is required. Since a causal block diagram

has values automatically propagated from the leaf blocks, blocks do not need to explicitly know about their children. In contrast, behaviour tree nodes need to query their children. However, since causal blocks store their parents, it would be relatively trivial to indicate which edges lead back to a child node. Formally, O (a node's outward links) = E (outward links for a block). Also, if $e \in E$ leads to node N_i, then edge e can be added to I in node N. In this way, nodes can be made aware of their children. Note that care must be taken to order the children in a consistent manner. As seen above, sequence nodes and selector nodes both require their children to be queried in a particular manner as desired by the tree designer. Another note is that the values for Y are different. In causal blocks, values are over $\mathbb{R}$, while behaviour trees have $Y \in \{0, 1\}$. This issue may prevent causal blocks from being embedded in regular behaviour trees. However, this issue is handled in the hybrid formalism presented here, as discussed below in the design section.

3.3.2. Node to Block Mapping

To transform a node back into a block, X, Y are equal for both formalisms. O for a node becomes E for a block, while I is discarded. The function F can also be transformed, with a caveat. If the node's F performed global actions, these may require significant changes in order to fit within the causal block formalism. For instance, a node's function may play an animation, or refer to another agent in the environment. Theoretically, these actions can be modelled as causal block diagrams. An animation could be adding constants to bones in the agent's model, which changes their position. Other agents could be modelled as their own causal block diagram, with values propagating when required. In practice, this lack of global state means that this transition may be highly impractical. The case is further complicated when the number of agents changes during a simulation. In behaviour trees, the global state would accomodate these new agents. In a causal block diagram, whole new diagrams would have to be grafted in and out of a model.

Care must also be taken for the sequence and selector node. While their function could be represented by an unwieldly cascade of test blocks, it is not immediately apparent how this process could be randomized to implement random sequencers and random selectors.

4. Design

This section will detail how some of the causal blocks were implemented as behaviour tree nodes, as well as the construction process of the trees. Examples will be selected from the two experiments done.

4.1. Embedding Causal Blocks

The formal notations shown above suggests that a transformation procedure from causal blocks to behaviour tree nodes is fairly trivial to perform. It can be accomplished by defining nodes that have a one-to-one mapping onto causal blocks. One special consideration to note is that the return value for these nodes is now $Y \in \mathbb{R}$, similar to causal block diagrams. This allows the new hybrid trees to handle any value output from a node. As an example, the Adder block sums up all the X at the time step to produce Y. This is easily implemented in a behaviour tree node, with code similar to that shown below.

```
Multiplier node update():
if children.size == 0
   output = 0
Result = 1
   For Child c in children
   Result *= c.update()
Output = result
```

This code will be called when the Multiplier node is updated. It will simply return the product of its children's values. If there are no children, 0 is returned. This can be seen as an error code in the hybrid tree, similar to the false value in behaviour trees. The semantics of this error code are left up to the designer of the hybrid tree to handle. Also, it is worth noting here that no order is defined for the children. However, an order may be imposed by the tree designer, using the same code structure that maintains children ordering for the sequence and selector nodes. When the value is returned to the parent, a copy of the value is also kept in the node with a timestamp. If the node is updated again, it will simply return the stored value. In this way, repeat calculations are avoided, and the same semantics as causal blocks is preserved.

The bottom five nodes are of special interest. The Constant node holds a single value. Whenever it is queried, it returns this value, thereby querying

- Adder
- Multiplier
- Inverter
- Negator
- Constant
- Test
- Delay
- Integrator/Derivative

Figure 3: A full list of causal blocks implemented in the hybrid formalism

no children. The Test node updates a particular test child. If the value returned is 0 or more, than a particular child is updated. Likewise if the value was false. These 'true'/'false' children are attached to particular ports on the node. This allows the Test node to control the flow of updating, and is very similar to the implementation of causal block diagrams in AtoM3 from (Posse et al., 2002).

The Delay node queries its children as normal when updated. However, instead of returning this value, it returns the value from the last time-slice. This is equivalent to the semantics of the Delay block in causal block diagrams, and is easily implemented in this hybrid formalism, despite the differences with regular behaviour trees.

Using the causal block formalism, it can be shown that Integrator and Derivative blocks can be created through connecting the other blocks. This is also true with the embedded causal blocks. One of the experiments done in this paper was to demonstrate that an Integrator tree could be created and visualized. The results are shown below. Therefore, it is accurate to say that the semantics of the Integrator and Derivative blocks from causal block diagrams can be easily reproduced in this new hybrid formalism.

4.2. Building Hybrid Trees

With the new hybrid tree nodes implemented as above, the construction of a hybrid tree is straightforward. Each node has an addChild method to

quickly add a child to either a default or specified port. The specified port is for those nodes like the Test node, which must have a way of referring to the 'test', 'true' or 'false' child. The default port is used for other nodes such as the Multiplier, where the children can be placed in an unnamed list.

Figure 4 shows a sample embedding of causal blocks into a behaviour tree. Note that this is an artificial example and does not represent a real system. However, it does show the visualization of the tree created through a Python script, as well as combinations of nodes that may be used in a real system. Note that the output from add1 is directed to neg. While it may seem that this causes a cycle to form, the Delay node in the middle of the tree blocks the same-cycle dependency. More information on these loops may be found in Vangheluwe (2012).

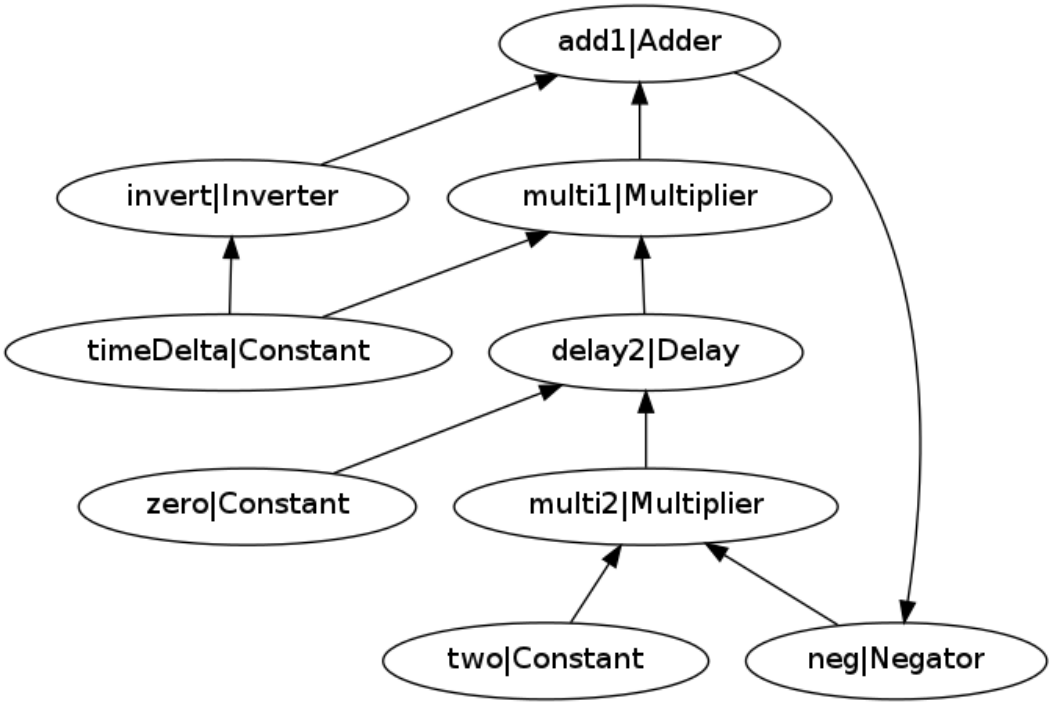

Figure 4: A sample hybrid tree diagram

4.2.1. Java Implementation

Example Java code to actually construct hybrid trees is given in figure 5. Note that a Random node is similar to a constant node, but produces a

random value every time-slice, where the value is chosen uniformally over the range between the two arguments given. The syntax to construct these trees is not the most elegant, but was simple and powerful enough to successfully perform two experiments as detailed below. The first parameter for each node is the parent node. The second is the port name as a String. For example, this may be Delay.`INPUT_PORT` to instruct the program to add this node to the Delay's node input port. As implemented, there is only minor error checking for erroneous port names, clobbering other children, and other errors. With a null parameter, the node is added to the default child list in a parent node. The third parameter is the name which is mainly used for debugging. Notice that the look and path nodes have a number in front of their name. This is so the root node, being a Sequence node, executes in the order specified.

```
Node root = new Sequence(null, null, "root");
    ActionLook look = new ActionLook(root, null, "1look");

    ActionMove randomMove =
        new ActionMove(root, null, "2randomMove");

        float randAmnt = 0.3f;
        RandomValue randVal1 =
            new RandomValue("randVal1", -randAmnt, randAmnt);
        randomMove.addChild(ActionMove.X_IN, randVal1);

        RandomValue randVal2 =
            new RandomValue("randVal2", -randAmnt, randAmnt);
        randomMove.addChild(ActionMove.Y_IN, randVal2);
```

Figure 5: Sample Java code to create a hybrid tree

5. Experiments

Using the techniques in the previous section, various hybrid trees were created. This section details two experiments performed with these hybrid trees in order to demonstrate the flexibility of the formalism, the visualization capabilities, and the numeric accuracy of a hybrid tree implementation compared to a causal block diagram implementation.

5.1. Circle Test

The circle test is an experiment to determine the numerical accuracy of the hybrid tree. The task is to simulate $d^2x/dt^2 = -x$ with $x(0) = 0$ and $dx/dt(0) = 1$. If $x(t)$ is plotted versus $dx/dt(t)$ for increasing values of t, the resulting plot should show a circle. The experiment was first performed using a causal block diagram in order to obtain numerical results. The causal block diagram can be seen in figure 6, with the resulting plot in figure 7. Note that the integrator block is an abstract one, made up of other blocks. Due to a fixed time step, this formalism cannot accurately represent truly continuous systems, and the system diverges from the true circle over time.

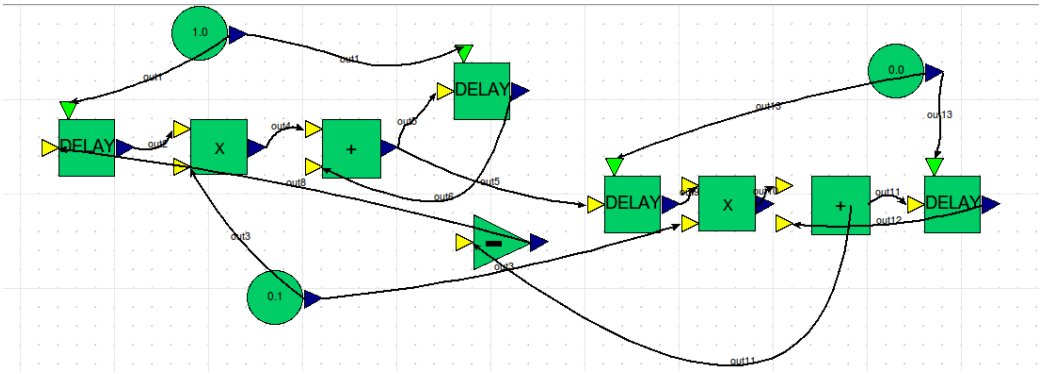

Figure 6: The circle test as implemented in a causal block diagram

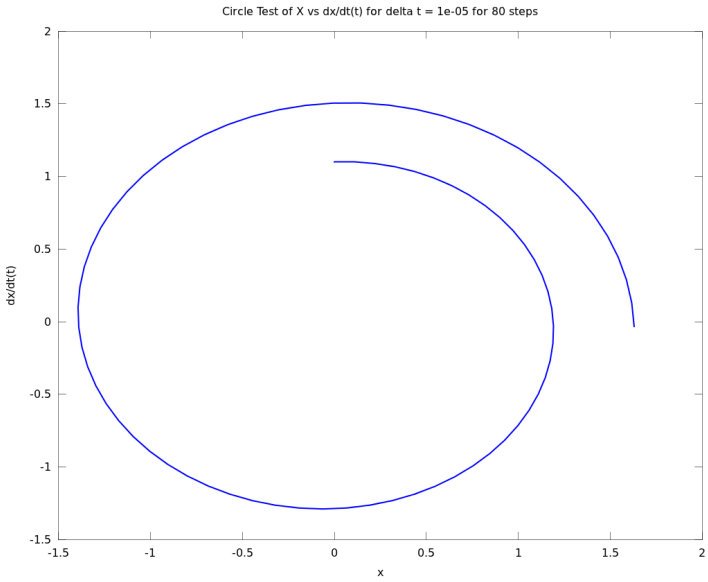

Figure 7: x(t) plotted versus dx/dt(t) for the causal block diagram

The same model was then faithfully reproduced in the hybrid tree formalism. The resulting hybrid tree can be seen in figure 8, with a moment in the simulation captured in figure 9. Notice that the exact same divergence is also seen in this model.

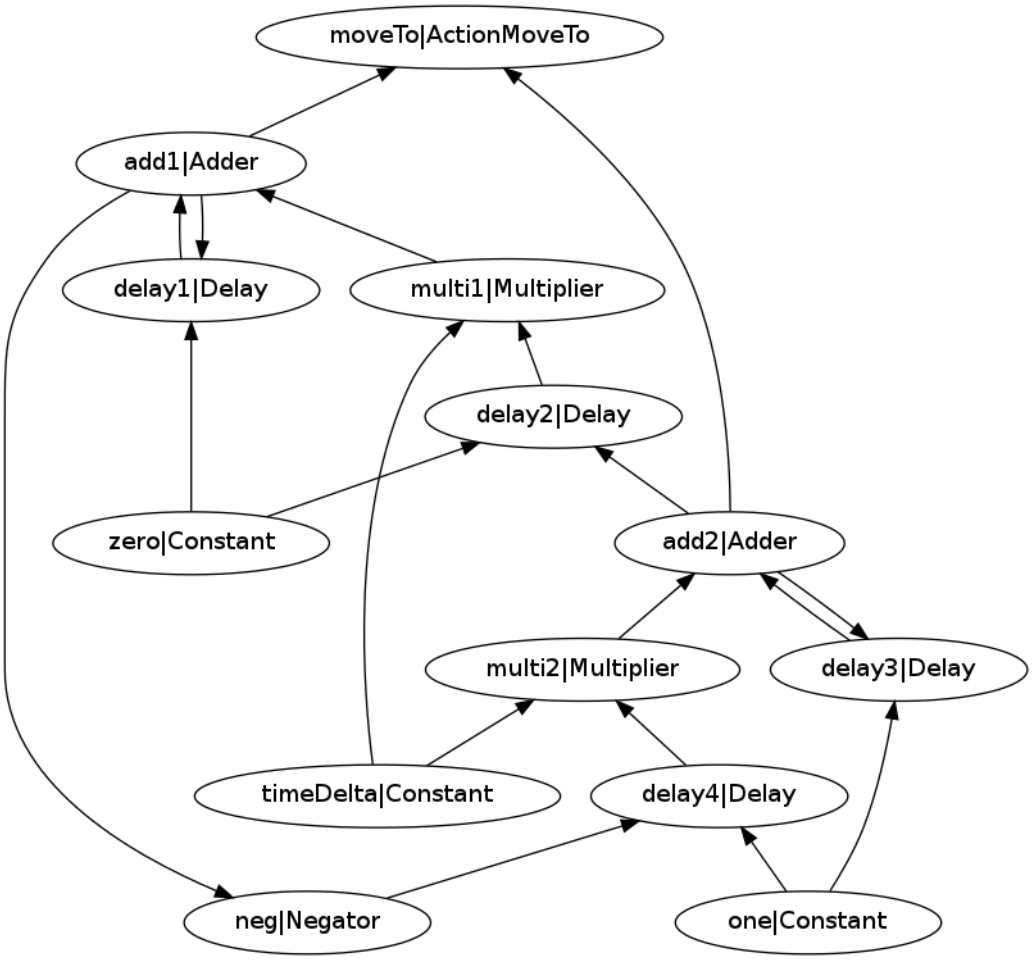

Figure 8: A hybrid tree for the circle test

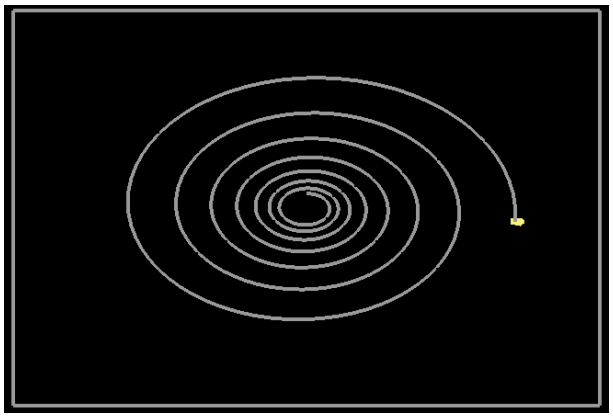

Figure 9: The circle test simulation after a few seconds

These final results are the output for the circle tests in both tested formalisms. To a high degree of precision, the values produced were exactly the same (the output has been simplified for the sake of clarity). Therefore, the hybrid tree is an exact transformation of the causal block diagram. However, this output can be directly visualized in real-time by connecting the hybrid tree to a simulation as done above. This allows the output of a model to be seen evolving over time, which may be important for some systems.

```
Causal Block Diagram Output:
add0 Adder
[1.1, 1.1, 1.089, 1.067, 1.034, 0.990, 0.936, 0.872, 0.799, 0.717]
add1 Adder
[0.0, 0.110, 0.220, 0.328, 0.435, 0.539, 0.638 0.731, 0.819, 0.898]

Hybrid Formalism Output:
Node: add2
    0     1     2     3     4     5     6     7     8     9
1.100 1.100 1.089 1.067 1.034 0.990 0.936 0.872 0.799 0.717
Node: add1
    0     1     2     3     4     5     6     7     8     9
0.000 0.110 0.220 0.328 0.435 0.539 0.638 0.731 0.819 0.898
```

5.2. Personal Space Simulation

The other experiment that was performed was to model the personal space requirements of an agent as it moved through a space crowded with

other agents. This was done in order to test a larger tree, as well as nodes that obtained information from the global simulation state. The test involved multiple agents divided into two categories: a walker and non-walkers. The walker has the task of moving from a start position to the goal position. However, both the walker and the non-walkers were also programmed to move away from others encroaching on their personal space.

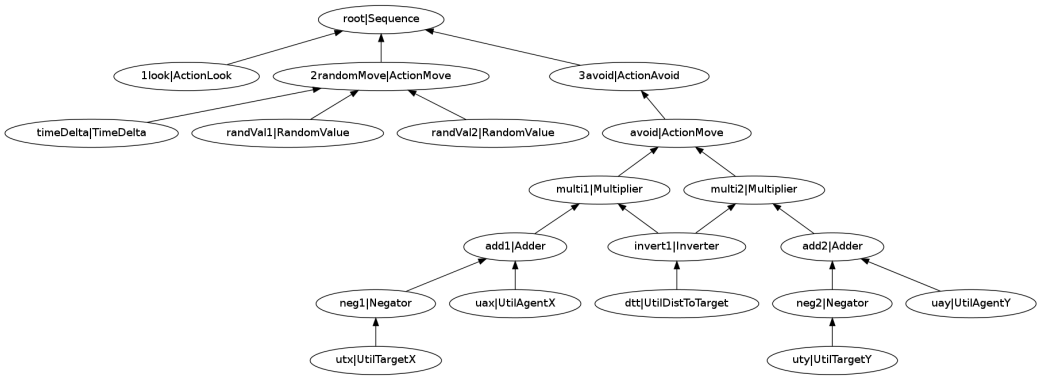

Figure 10: The hybrid tree for the non-walker agents

Figure 10 shows the hybrid tree created to implement the agent-avoiding behaviour for the non-walker agents. The look node on the left maintains a list of agents close to this agent. It performs this action by calling a method referencing the global state. This is allowed by the behaviour tree formalism, and gives the hybrid much more power than a related causal block diagram (as mentioned above, however, there may be a causal block diagram that matches this hybrid tree). The use of global knowledge is also used in the leaf nodes of the tree, which returns the distance between the agent and their target. This target is selected by the ActionAvoid node, which repeatedly sets the target to a different agent close in proximity and update the move action. The effect of these looping motion is to move the agent away from all nearby agents. This is done using a simple potential field calculation, summing up all forces acting on the agent. In this scenario, the force is a repulsive one away from close agents, modulated by distance. Note that this loop and summing function could be expressed in an extremely large causal block diagram. However, the use of this hybrid tree allows the representation to be relatively compact.

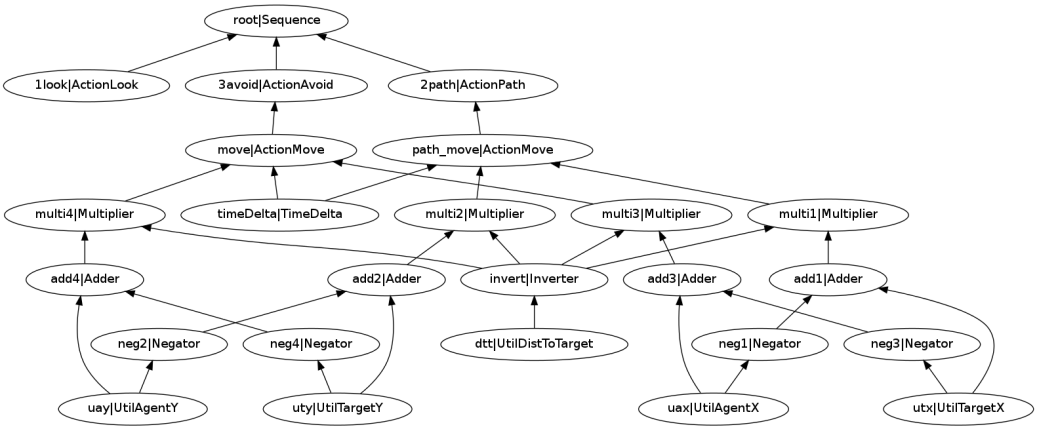

Figure 11: The hybrid test for the walker agent

The hybrid tree for the walker agent is shown in figure 11. It is similar to the non-walker hybrid tree, but adds a force to pull the walker agent towards the goal position. This goal position is set to be the target of the agent by the ActionPath node. This allows the bottom part of the tree to be shared between the avoid behaviour and the path action, reducing code complexity.

The simulation is shown in figure 12. The walker agent is the gold square in the lower-left, while the non-walker agents are spread randomly through the space and are represented in blue. Notice that the non-walkers have already spread themselves out relatively uniformly throughout the space. This is due to their avoid behaviour. The walker is trying to move through the non-walkers, from the green start square in the extreme lower-left to the red goal square in the extreme upper-right.

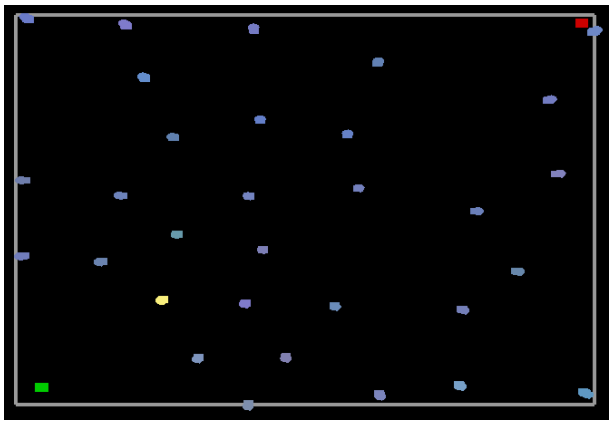

Figure 12: The personal space simulation

5.3. Using Hybrid Trees

In this paper, these hybrid trees were implemented as source code statements in Java. This meant that there was no form of visual construction. A complicated method of node relationships led to errors, as typos in the code caused undesirable behaviours. However, a Python program was developed to read the hybrid tree specification and produce a tree diagram. This allowed work to proceed smoothly, as nodes that had been wrongly connected could be easily identified. In fact, the experiment involving the circle test was coded and performed in less than a day. This speed was due to the rapid construction and iteration allowed by the software created.

The flexibility of Java allowed the hybrid trees to easily access any information in the world for use in computation. This was instrumental in quickly enabling agents in the personal space simulator to move away from each other. Furthermore, the Java OpenGL implementation provided an attractive visualization for the hybrid tree's results. This visualization could easily switch between the two scenarios presented in this paper, and could easily accomodate other experiments.

6. Comparison with Other Works

This paper has taken the unique approach of embedding another formalism within behaviour trees. In the related literature, there are few examples to compare this paper's work to. The brief formalism descriptions above have attempted to merge Colvin and Hayes (2007) and Denckla and Mosterman

(2005), with some success in producing a hybrid formalism. However, the focus of this paper was not on the formalism descriptions, and as such is less detailed than the work of others. Liu and Baltes (2004) has interesting ideas about the flexibility of behaviour trees for the control of a robotic soccer team. The work done in this paper shows that other formalisms, specifically causal block diagrams, may be used in order to increase the performance of agents. Models may be tested in other environments and then easily embedded into a hybrid formalism.

The work of Posse et al. (2002) and the lectures of Vangheluwe (2012) show a need for visualization of causal block diagrams, which has been attempted in AtoM3. However, as discussed above, a textual representation of the causal block diagram's output may not be suitable for some problems. For example, the output for the personal space simulation would be absolutely incomprehensible in any form other than in simulation. The hybrid formalism described in this paper may be connected to a simulation, allowing the output of the causal block to be visualized over time as the system evolves. The power and compactness found in behaviour trees is also transferred to the hybrid formalism, allowing for systems that may be impractical in a causal block diagram.

7. Conclusion

This section of the paper consists of two subsections. The first will detail future work related to these hybrid trees, while the second will offer a recap of the paper as well as some concluding remarks.

7.1. Future Work

This section will detail three areas in which this hybrid model may be extended, as well as descriptions of how it may be integrated into modelling tools.

7.1.1. Algebraic Loops

When a causal block diagram is being simulated, dependencies may occur. This is when one block depends on an input value from a second block, which in turn depends on the first block (possibly through a long chain of blocks). Therefore, at any particular time-slice this deadlock may occur. This can be handled in causal block diagrams through a strong component solver, as seen in the first assignment in Vangheluwe (2012). A linear solver is then used to

solve the algebraic loop if possible. In some cases, the loop cannot be solved, and the system is in error.

In the above implementation, there is an assumption that there are no strong components. Notice that there may be loops, as in the circle test. However, these loops are not strong components, as the Delay block does not depend on the input value at the current time-slice, but instead the value at the previous time-slice. A loop solver may be feasible to include in the hybrid formalism described. However, the performance and complexity of the implementation would grow significantly.

7.1.2. AtoM3

AtoM3 is a modelling tool as described in Posse et al. (2002), which enables causal block diagrams (among others), to be created easily. These models can then be synthesized into Python code to be run in a simulator. The AtoM3 software offers a number of consistency checks, as well as tools to combine formalisms in one diagram.

It may be advantageous to model the above hybrid formalism within AtoM3 in order to work within a professional modelling environment. Writing the Java code to link together nodes and then run the Python graphing code may be a slower and more erroneous process than directly linking nodes in AtoM3.

7.1.3. Automatic Transformations

One further area of future work would be to explore the automatic transformation of a causal block diagram to and from a hybrid tree. This work would entail precisely formulating a transformation operation, which would build upon the formal equivalence work as touched on in this paper, Denckla and Mosterman (2005), and Colvin and Hayes (2007). The difficulties as mentioned above would require significant time and thought to resolve, as it does not appear to be a one-to-one mapping from behaviour trees to causal block diagrams. However, a transformation operation would allow models from one type of formalism to be automatically transformed into the other. Advantages of this may be a more compact representation, or an efficient simulator may be available for only one formalism.

7.2. Final Discussion

This paper has presented a novel hybrid formalism of causal blocks embedded within behaviour trees. This was done in order to leverage the power

of behaviour trees in such areas as control flow, flexibility, and visualization. The embedding of the causal blocks was described, as well as the changes to the behaviour tree formalism that had to be made to accomodate the causal blocks. Two models were presented. The first model was a circle test to test the produced results against an implementation in causal blocks, as well as demonstrate the evolution of the model in a simulation. The second experiment was to model an agent walking through a crowded room. The agents each had their own hybrid tree in order to control their movement.

The combination of causal blocks and behaviour trees is a natural one. The hybrid tree, as described above, offers improved performance over causal blocks, as well as offering an opportunity to simulate the results of the causal blocks. Behaviour trees also bring a robust flexibility which allows powerful new nodes to be created, as well as the capability to rearrange tree nodes at run time. By combining these formalisms, it is possible to develop causal block diagrams with added power. These diagrams can then be simulated, allowing rapid iteration of the model layout to achieve desired results.

8. References

Colvin, R., Hayes, I. J., 2007. A Semantics for Behavior Trees. Tech. rep., ARC Centre for Complex Systems.

Denckla, B., Mosterman, P. J., 2005. Formalizing Causal Block Diagrams for Modeling a Class of Hybrid Dynamic Systems, 2005.

Hecker, C., 2005. My Liner Notes for Spore/Spore Behavior Tree Docs, `http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs`. Accessed April 26, 2012.

Isla, D., 2005. Managing Complexity in the Halo 2 AI System, `http://www.gamasutra.com/view/feature/2250/gdc_2005_proceeding_handling_.php`. Accessed April 26, 2012.

Lim, C.-U., June 2009. An A.I. Player for DEFCON: An Evolutionary Approach Using Behavior Trees, Imperial College, London.

Liu, X.-W. T., Baltes, J., 2004. An Intuitive and Flexible Architecture for Intelligent Mobile Robots. In: 2nd International Conference on Autonomous Robots and Agents.

Posse, E., de Lara, J., Vangheluwe, H., April 2002. Processing Causal Block Diagrams with Graph-grammars in AToM3. In: European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT). pp. 23 – 34.

Vangheluwe, H., 2012. COMP 522 - Modelling and Simulation course notes, `http://msdl.cs.mcgill.ca/people/hv/teaching/MS/lectures/`. Accessed April 26, 2012.