Behavior Trees for UAV Mission Management

Andreas Klöckner

Institute of System Dynamics and Control
German Aerospace Center
Münchner Str. 20
82234 Oberpfaffenhofen, Germany
andreas.kloeckner@dlr.de

Abstract: Behavior trees are a recent development in artificial intelligence for computer games. Their application has been proposed to increase modularity of an unmanned aerial vehicle's control system. This paper first describes advantages of behavior trees for use in mission management. The paper then points out research towards their deployment. Finally, newly developed transient tasks are introduced to behavior trees. These allow for finite-state-machine-like initialization and interruption of tasks. The use of the system is demonstrated with an unmanned aircraft mission simulation.

1 Introduction

There is a general consensus that unmanned aerial vehicles (UAVs) need to be provided with more autonomy in order for them to effectively fulfill industrially relevant missions. Increasing a UAV's autonomy reduces the workload for the ground crew and eventually will allow for an operator-to-system ratio of less than one. This ability of UAVs to operate more or less unattendedly will make their use more profitable for industrial purposes.

However, current research in UAV autonomy concentrates mainly on developing disjointed capabilities. There have e.g. been substantial advances in UAV control, path planning, collision avoidance, formation flying or physical interaction with the environment.

As missions grow more complex, these capabilities need to be integrated. A similar problem is encountered in the computer game industry, when behaviors of complex non-player characters have to be integrated. Conventional approaches to the problem include e.g. finite state machines, scripting and hierarchical task network planning. Artificial intelligence programmers have recently established behavior trees as a new approach, which combines several advantages of conventional approaches [Isl05, Cha07]. An introductory description of the technique can be found in Millington's textbook [MF09].

In this paper, Ögren's proposal [Ögr12] to use behavior trees for UAV control is picked up. First, the formalism of behavior trees is introduced briefly in Sec. 2. Section 3 discusses advantages of the formalism for mission management. In Sec. 4, the context of this paper in a broader research field is outlined. This paper's main contribution of introducing transient tasks to behavior trees is described in Sec. 5 and the use of the enhanced system is demonstrated in Sec. 6. A brief conclusion finally is provided in Sec. 7.

2 Behavior Trees

Behavior Trees (BTs) are similar to Hierarchical Finite State Machines (HFSMs) in that they build on a hierarchy of operational modes. While the state transitions of HFSMs are triggered by discrete events, behavior trees build on tasks that are enabled and disabled by persisting statuses.

A *task* is a self-contained goal-directed behavior, which can be executed in order to achieve a given goal. One of the simplest tasks within a UAV mission management system could be to engage a specific autopilot mode, e.g. flight level change. By combining tasks to composite tasks in a tree structure, behaviors of arbitrary complexity can be generated.

All tasks have the same simple interface: They can be *ticked* to execute them. A complete behavior tree is executed by ticking its top-most (root) task. The ticks are then propagated through the composite nodes to the atomic tasks at the tree's leaf nodes, which actually carry out actions on the environment. When ticked, a task returns one of three status return codes: It can be finished with *Success* (✓), aborted in a *Failure* (↯) or be *Running* (▶).

From this basic notion of a task, two atomic task types can be derived: Actions and conditions. While these atomic tasks determine their status based on the custom implementation they are provided with, composite tasks determine their own status as a specific function of their sub-tasks' statuses. The basic composite tasks of behavior trees are sequences and selectors. The four basic task types are characterized as follows:

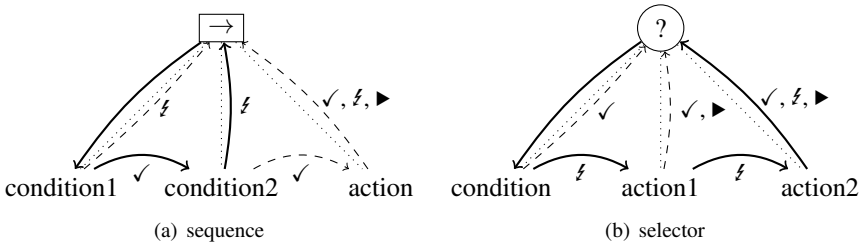
Actions are pieces of custom code that are intended to alter the system's environment, such as engaging autopilot modes, switching payloads, or taking photographs.

Conditions are specialized actions, which do not alter the environment, but test for some properties of it. They cannot have a running status, but immediately return Success or Failure on a tick.

Sequences tick each of their sub-tasks in order and return success, when all of them are successfully finished. A sequence can be regarded as a goal-directed execution of a series of tasks. When compared to logic, the sequence draws an analogy to the AND operator.

Selectors are the complement to a sequence. They also tick each of their sub-tasks and return success, when either one of their sub-tasks is finished with success. A selector is thus a list of prioritized actions, one of which will be chosen for execution dynamically. The selector is the correspondence to a logical OR.

Figure 1 depicts the execution of the standard composite tasks during one tick. The sequence in Fig. 1(a) can be regarded as executing the action only, if both pre-conditions are satisfied. Thick arrows illustrate the processing for the case, that the second pre-condition is not satisfied. The selector in Fig. 1(b) can be regarded as two alternative actions serving a common purpose, namely to render the condition true. Thick arrows illustrate the case, that the first action fails in doing so and the alternative action2 is currently running.



(a) sequence (b) selector

Figure 1: Processing of the basic composite tasks during one tick

A multitude of additional composite tasks can be imagined: Parallel tasks can execute several sub-tasks simultaneously. Decorators can be used to alter a single sub-task's return code or to guard shared resources. Loops can be used as specialized decorators for repeating the same sub-task multiple times. For more detailed information on behavior trees, their implementation and common extensions, please refer to Millington's textbook [MF09].

3 Prospects of Behavior Trees for Mission Management

The state-of-the-art approach for managing different capabilities of a UAV is to use Finite State Machines (FSMs). However, with an increasing number of capabilities, FSMs do not scale very well and become difficult to maintain. Behavior Trees promise to remedy this shortcoming and to provide some additional desirable properties of a mission management system as argued in the following paragraphs.

In order for a mission management scheme to be scalable, i.e. to stay tractable with increasing complexity, the system should be flexible, i.e. easily changeable, possibly during execution. In particular, one expects the system to be modular: Changes in one part of the system should not influence a different part of the system. Behavior trees provide these properties by implicitly encoding transitions in their tree structure and by their standardized interfaces. Adding or removing nodes of a BT requires the modification of a single connection, whereas re-wiring the transitions of FSMs can be a tedious work. Obviously, modifying one branch of the tree will have a local effect only and will not affect other branches. The standardized interface ensures that it is possible to add and remove arbitrary branches of the tree.

The fact, that basic behavior trees do not have internal memory but rely on persisting statuses, additionally provides for run-time changes to a mission plan: While a considerable amount of work has to be invested to appropriately re-initialize FSMs after changes, BTs will automatically activate their appropriate branches.

Additionally, one would like the mission management scheme to be intuitive, such that it can be easily understood by an end-user. This is facilitated by the goal-directed nature of BTs. Each task can be thought of as trying to accomplish a particular sub-goal and the BT is built up by combining these tasks to achieve higher level goals. Additionally, the user

can arbitrarily work with re-usable, familiar sub-tasks (e.g. from a library) because of the standardized interfaces of BTs.

The tree structure of BTs and the close proximity of the basic composite nodes to the logical operators (AND/OR) seem additionally promising for use in validation and verification of aspects, such as whether all possible inputs are covered with a response action or whether a BT finishes successfully only, if a given goal has been achieved.

As lined out, BTs are described also in the literature to combine a number of positive factors from many fields such as scripting, planning and state machines [MF09, Cha07]. However, it has to be noted that BTs cannot solve every problem. For example, BTs as well as FSMs provide reactive schemes only, which do not plan ahead in order to foresee necessary actions in the future.

4 Context of this Work

The present work is part of an ongoing doctoral research project aiming to introduce behavior trees to UAV mission management. Current UAV control is often comprised of an autopilot stabilizing the UAV and a guidance loop dictating flight destination and trajectories. In the context of this project, the loop shall be supplemented with a mission management module pursuing overall mission goals as shown in Fig. 2. The mission management shall be able to use all the capabilities of the UAV, including payload operation.

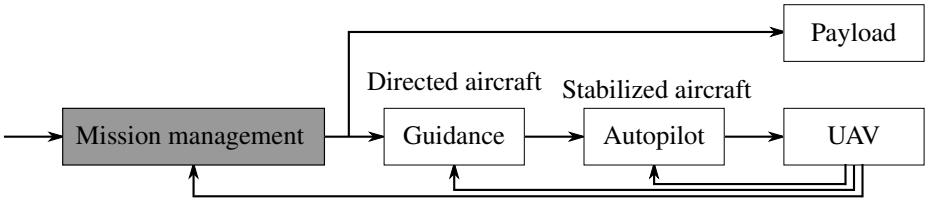


Figure 2: Location of the mission management within the UAV control loop

In order for a potential user to design efficient BT mission plans, they must be provided with specific *synthesis* tools. To that end, a library of available tasks is to be built up. In a mission design tool, these tasks shall be interfaced with simulation and optimization routines. Genetic algorithms have been shown to be usable on behavior trees in [LBC10]. It shall also be investigated, whether learning algorithms such as decision tree learning can be used to automatically build up a mission plan from the available tasks and specified goals.

Especially in aerospace applications, it is crucial to provide formal *analysis* of every component involved in the control loop. A major emphasis of the planned research is thus directed towards analyzing BTs. On the one hand, BTs will be thoroughly compared to similar methods such as state machines. It is expected that behavior trees will be translatable into these formalisms, making available existing analysis tools to detect inconsistent execution states or to generate suitable test cases for validation. Additionally, for real time deployment of behavior trees, the reaction times of a given tree must be guaranteed.

On the other hand, the close relation to logic shall be exploited in order to develop new algorithms for validation/verification. To that end, all atomic tasks shall be converted to logical formulas. These atomic formulas can be combined according to the decision rules of the composite tasks into a single logical statement. By combining this statement with assertions about the world, it will be possible to derive properties, e.g. that the BT always executes an action, or that the BT returns successfully if and only if the specified goal has been reached. With suitable UAV and world models, it will be investigated whether or not it is possible to further guarantee that a plan always reaches its goal. A first approach to integrating BTs with a description logic is outlined in [Klö13].

The third aspect of the research will deal with *implementation* issues of BTs. The final goal of the research will be to provide BT mission management to the operator of complex UAVs, such as described in [KLSL13]. A major integrative concern will be to provide the user with flexible means to design, analyze, change and upload BT mission plans to the UAV. This also includes efficient flight algorithms, e.g. based on [Kna11].

Extensions to the implementation will be provided as identified by the needs of applications. For example, typical control systems are expected to work with event-based switches. While the current behavior trees work on continuous statuses, events can be introduced either by providing memory to specialized tasks or, in line with the regular BT philosophy, by converting events to statuses at a lower level of the mission plan execution. These statuses can be stored in a blackboard mechanism and reset after processing by suitable BT tasks.

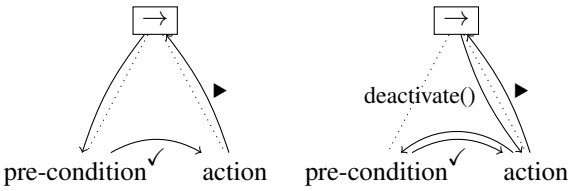
This paper describes one of the identified extensions: Providing tasks with entry and exit hooks by using transient behaviors. This extension narrows the gap between current BT features and state machine features expected by the user.

5 Transient Behaviors

A major shortcoming of the aforementioned implementation as compared to Finite State Machines is, that it does not provide for clean exit conditions of behaviors. In case a behavior should be properly interrupted by the superior nodes in the tree, it is simply not ticked anymore without notification. Suppose a remote sensing mission has to be aborted due to a system failure. One would wish to deactivate the power-consuming cameras during execution of the fallback behavior.

An elegant solution to this problem is to provide entry() and exit() function hooks as encountered in state machines. These are to be called upon activation and deactivation of the behavior respectively. A task must thus be provided with an internal status, which can at least be Idle or Running. If a currently Idle sub-task needs to be executed by its superior node, the currently Running sub-task needs to be deactivated first and the sub-task to be ticked needs to be activated.

However, a new problem is introduced, when behaviors are activated and deactivated unconditionally. The issue is illustrated in Fig. 3 with a conventional sequence composed of a pre-condition and an actual action. Figure 3(a) illustrates the first tick to the idle sequence node: First, the pre-condition is activated. Its tick evaluates to Success, such



(a) first tick (b) subsequent tick

Figure 3: Chattering introduced by activation and deactivation procedures

that the pre-condition is deactivated and the actual action is activated. The tick of the action then returns Running. Figure 3(b) shows a subsequent tick to the same sequence. It unconditionally starts by deactivating the action, in order to activate and tick the pre-condition. The pre-condition will again evaluate to Success. It will be deactivated and the actual action will be re-activated. In other words, chattering is introduced.

A way to counter this problem, is to use Hecker's [Hec09] approach to wrap the activation and deactivation of behaviors inside decider functions. These are called before the actual tick and decide for their sub-tasks, which if any of them to activate. This makes for explicit, custom-made deciding algorithms. Some substantial amount of work will have to be spent in order to guarantee consistency for all combinations of deciders. It must especially be guaranteed, that the activated sub-task will actually execute.

In the present work, a different, implicit approach to the problem is proposed by introducing the notion of *transient* behaviors. Transient behaviors will not enter running state, when ticked, and thus do not need to be activated. Conditions are the natural prototype of transient behaviors, because they can never be running. They are thus called explicitly transient. The notion can be generalized for arbitrary tasks, including composite tasks. A task is called implicitly transient, if it could potentially be activated but behaves transiently in response to the current tick.

For this approach to work, the current task status needs to be extended. Each task starts out with an Idle status. When it is ticked, it determines, whether to behave transiently or not: A transient task immediately returns either Success or Failure. Because a transient task does not enter running state, its status remains Idle after the tick. A non-transient task, however, will enter the new status Activating and also return an Activating code to the superior task. This can be understood as a request to be activated by the superior task.

The actual activation is performed through the superior composite task. It first deactivates a possibly running sub-task and then activates the requesting sub-task. The requesting sub-task enters the Running state during the activation. In order to allow to deactivate currently running tasks first, tasks are forbidden to self-activate.

While the Success and Failure return codes of transient tasks can be handled the same as before, for non-transient behaviors these return codes are additionally interpreted as a request for deactivation. A superior task will follow this request and possibly activate one of its other sub-tasks afterwards.

Table 1: Possible return codes of transient and non-transient tasks

	Possible return codes			
	Activating	Running	Success	Failure
Idle status	non-transient		transient	transient
Running status		non-transient	non-transient	non-transient

Table 1 lists the possible return codes of transient and non-transient tasks. It only lists return codes for the Idle and Running status, because a task is only ticked in one of these states. A composite task can determine, whether its sub-task is transient by comparing the allowed return codes against the actual return code and the sub-task's current status.

Only when activated or deactivated by the superior composite task can a sub-task enter Running or Idle status respectively. This means, an additional layer of hysteresis is introduced. It behaves similarly to the one proposed by Hecker, but avoids the additional decider function. It instead relies only on the tasks' statuses and thus keeps the simple and modular interface of a general task. Figure 4 depicts the resulting status cycle for non-transient tasks. Transient tasks will never change their internal status from being Idle.

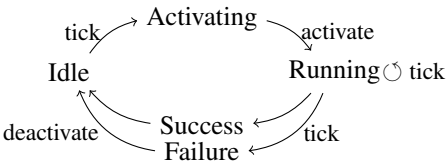


Figure 4: Status cycle for non-transient behaviors.

The activation and deactivation procedures now naturally provide the required entry and exit hooks for clean interruption of behaviors. By distributing the activation decision to the leaf tasks, it is naturally guaranteed, that an activated sub-task will accept its activation.

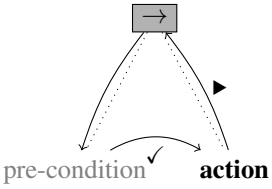
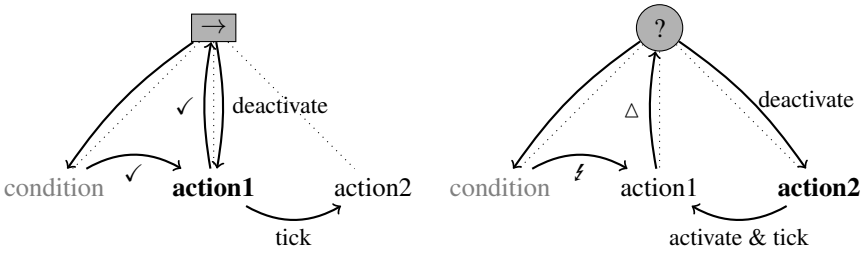


Figure 5: Tick to a running sequence task with a transient pre-condition

As transient tasks will never be formally activated, this also solves the chattering problem described earlier: The pre-condition is explicitly transient and will thus not request activation. Its tick will immediately return Success instead. The superior sequence task will then immediately proceed to the actual action without interrupting it in the meantime. The tick of the action will return Running again. See Fig. 5 for an illustration. The extended

composite task is represented by a gray filling and the explicitly transient condition is grayed out. The currently running sub-task is highlighted in bold font.

The modified composite tasks will now only deactivate a currently running sub-task, if it requests to be deactivated by returning Success or Failure, or if a sub-task of higher priority behaves non-transiently and requests activation. The first case is depicted in Fig. 6(a). The currently running sub-task is action1. When the sequence is ticked, it starts with its first sub-task, which behaves transiently and returns Success. The tick to the currently running action1 returns Success. It thus prompts the sequence to deactivate it. Subsequently, the next sub-task (action2) is ticked.



(a) sequence with a finishing sub-task (b) selector activating a sub-task of higher priority

Figure 6: Processing of the modified composite tasks during one tick

The case of a higher priority sub-task requesting activation is depicted in Fig. 6(b). The modified selector starts by ticking its first sub-task, which transiently returns Failure. The second sub-task, however, returns Activating (△). The selector thus deactivates its currently running sub-task action2 and activates the higher priority sub-task action1. It then re-ticks action1, in order to retrieve a valid status to return to its superior node. Note that this re-tick could also result in a Failure or Success return code, which would issue the selector to re-activate action2 or to return control to its superior task respectively.

Note also, that currently a composite task must follow a child's request for activation. This behavior can be extended to implement more elaborate composite nodes, which can choose from a set of possibly activating sub-tasks. This allows for example to choose from a selector's sub-tasks based on dynamic priorities such as described in [Dyc07].

6 Demonstration of Application

The described system is employed for a test case. A point mass model is used to represent the aircraft together with its autopilot. The model is steered by the behavior tree with a mission profile as shown in Fig. 7. The plan describes a take-off (T/O) procedure and a repeated set of waypoints to be flown subsequently.

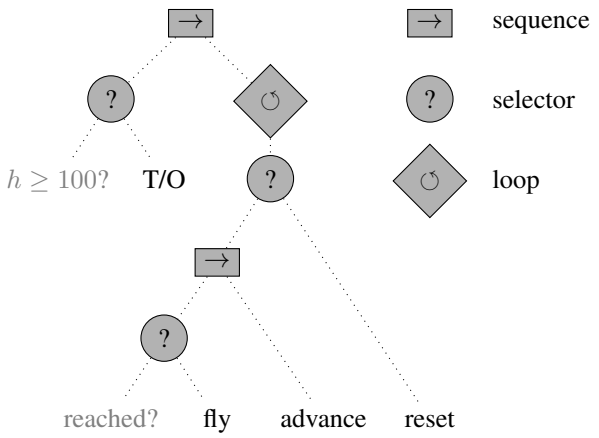


Figure 7: A simple mission profile for take-off and traffic circuits

The objective of the first part of the plan is to reach a given altitude. It is explicitly introduced within a selector node. Thus, if the plan is activated in the flying aircraft, no take-off is executed. Different strategies to gain altitude could be introduced as additional sub-tasks of this first selector node.

If a sufficient altitude is reached, the plan switches to its waypoint-following part. The central task is to "fly" to the current waypoint. If the waypoint is "reached", the autopilot's internal waypoint pointer is "advanced". If the advance task fails, the internal pointer is "reset" to the first waypoint. The complete procedure is repeatedly executed by a loop task.

The evolution of the behavior tree's statuses is depicted in Fig. 8 for all leaf nodes. The different statuses are shown as Idle ($\cdot$), Activating ($\triangle$), Running ($\blacktriangleright$), Success (+) and Failure ($\circ$). They are shown on different levels of the y axes for clarity. The figure shows results from a simulation with a take-off, a full waypoint cycle and a reset of the waypoints.

All tasks start off in Idle state. In the beginning of the simulation, the altitude check fails and the take-off task is activated. When the altitude check evaluates to Success after about $100\,\mathrm{s}$, the take-off task is deactivated and returns to Idle status. It is then checked, if the first waypoint has been reached. The check fails, such that the aircraft is steered towards the first waypoint.

After about $120\,\mathrm{s}$, the first waypoint is reached. The "fly" action is deactivated first, then the "advance" action is ticked. It returns Activating and is thus activated, advancing the autopilot's waypoint pointer. It is then ticked again, allowing it to return Success immediately. The advance action is deactivated and the success is propagated up to the loop node, which in turn issues a new tick of the complete waypoint plan. The new current waypoint is now not reached and the "fly" action is re-activated.

All these status changes are iterated at the same time step of the simulation, allowing for the instantaneous "advance" action to be evaluated immediately, while the "fly" action is running throughout the continuous simulation. Instantaneous actions like the "advance" action can also be implemented as transient tasks, reducing the need for task (de-)activation.

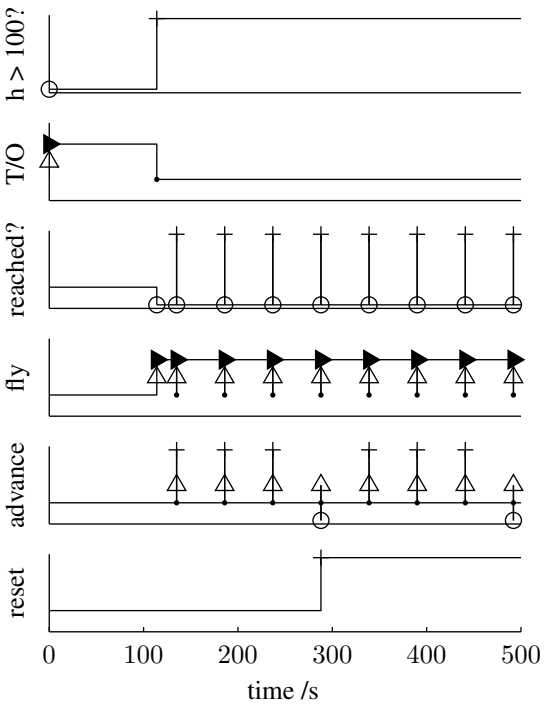
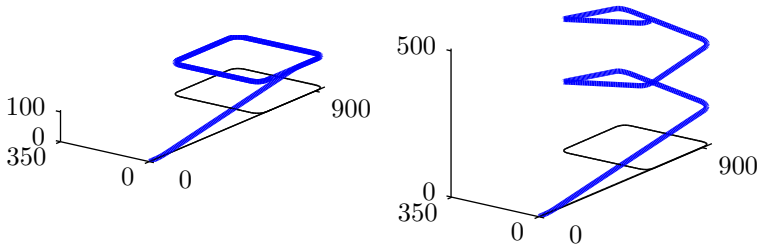


Figure 8: Evaluation of the exemplary mission profile

When the "advance" action is activated for the fourth time, it returns Failure. This indicates that the last waypoint is reached and the waypoint pointer cannot be advanced. The "reset" action thus activates, returns "Success" and triggers the loop to restart the waypoint plan.

Central enabler for this type of plan is the clean exit condition of the take-off task. When it is deactivated, the take-off task resets the autopilot to a safe altitude and speed hold mode with its exit hook. This prevents the aircraft from constantly climbing after the take-off as shown by the trajectories in Fig. 9. The same behavior could be achieved by chaining an explicit action for the exit conditions in a sequence to the take-off plan. This would, however, complicate the tree and reduce re-usability of the take-off task.

The example also shows the necessity for transient behaviors. Fig. 10 shows the evolution of the task states, if the altitude condition is modified to behave non-transiently. It is then activated and deactivated in each time step in turn with the actual take-off action.



(a) clean exit (b) no exit hook

Figure 9: Flight trajectories with and without exit hooks

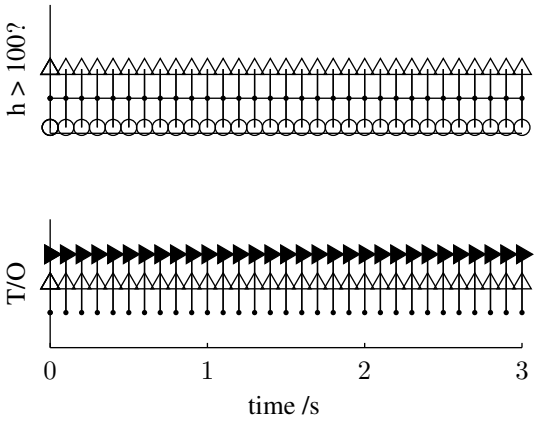


Figure 10: Chattering introduced without the new transient tasks

7 Conclusions

The powerful tool of behavior trees as developed in the computer game industry has been introduced briefly. It has been argued that behavior trees will prove advantageous for the problem of UAV mission management. As compared to the state-of-the-art solution using finite state machines, behavior trees offer increased scalability due to their tree structure and their simple, standardized interface. The goal-directed nature of their atomic tasks also provides a more intuitive framework to end-users.

Research directions have been laid out aiming to introduce behavior trees to UAV mission management by providing means for synthesis, analysis and implementation. This includes a library of UAV capabilities and tools to automatically build mission plans. Plans will be analyzed by comparing behavior trees to related formalisms and by exploiting their close relation to logic. Implementation will be addressed according to the applications' needs.

In this paper, entry and exit hooks have been added to the behavior tree formalism, in order to allow for guaranteed initialization and cleanup procedures of the tasks. The new problem of chattering is remedied by introducing the new notion of transient and non-transient tasks. This also comes with a new understanding of a tasks's status. The use of the modified behavior trees has been demonstrated at hand of a UAV mission example.

In summary, it has been shown that behavior trees can be used for UAV mission management. A few features will still have to be developed for online deployment to unmanned aircraft, such as event-based behavior trees. Approaches to these features have been identified. Behavior trees promise to be analyzable by traditional methods and additionally to be subject to further validation and verification algorithms. An approach to new algorithms has been lined out as combining behavior trees with logic.

References

[Cha07] Alex J. Champandard. Behavior Trees for Next-Gen Game AI. In *Game Developers Conference*, Lyon, France, 3-4 Decembre 2007.

[Dyc07] Max Dyckhoff. Decision making and knowledge representation in Halo 3. In *Machine Learning and Games (MALAGA) Workshop, NIPS'07*, 2007.

[Hec09] Chris Hecker. Spore Behavior Tree Docs. Technical report, Maxis / Electronic Arts, 12 April 2009.

[Isl05] Damian Isla. Handling complexity in the Halo 2 AI. In *Game Developers Conference*, 2005.

[Klö13] Andreas Klöckner. Interfacing Behavior Trees with the World Using Description Logic. In *AIAA Guidance, Navigation and Control Conference*, Boston, MA, 2013. AIAA. Accepted for publication.

[KLSL13] Andreas Klöckner, Martin Leitner, Daniel Schlabe, and Gertjan Looye. Integrated Modelling of an Unmanned High-Altitude Solar-Powered Aircraft for Control Law Design Analysis. In Qiping Chu, Bob Mulder, Daniel Choukroun, Erik-Jan van Kampen, Coen de Visser, and Gertjan Looye, editors, *CEAS EuroGNC*, volume Advances in Aerospace Guidance Navigation and Control, pages 535–548, Delft, The Netherlands, 10-12 April 2013 2013. Springer.

[Kna11] Björn Knafla. Data-Oriented Streams Spring Behavior Trees. Technical report, Codeplay, May 01 2011.

[LBC10] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game DEFCON. In *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 100–110. Springer, April 09 2010.

[MF09] Ian Millington and John Funge. *Artificial intelligence for games*. Morgan Kaufmann, Burlington, MA, 2nd edition, 2009.

[Ögr12] Petter Ögren. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. In *AIAA Guidance, Navigation and Control Conference*, Minneapolis, Minnesota, 13 - 16 August 2012. AIAA. AIAA 2012-4458.