The Modelica BehaviorTrees Library: Mission Planning in Continuous-Time for Unmanned Aircraft

Andreas Klöckner
German Aerospace Center (DLR), Institute of System Dynamics and Control
82234 Weßling, Germany, Andreas.Kloeckner@dlr.de

Abstract

The paper introduces a continuous-time architecture and a Modelica library for mission planning based on behavior trees. It allows to study the long-time behavior of complex aircraft models in interaction with reactive mission plans by means of efficient simulations. The developed Modelica library is used in a mission example for a solar high-altitude aircraft and the advantages of the behavior tree formulation in both simulation speed and modularity are discussed. The architecture will further be used to deploy automatically coded mission plans to actual flight computers using the functional mockup interface.

Keywords: UAV; Mission Management; Behavior Trees; Autonomy; Artificial Intelligence

1 Introduction

Missions currently envisaged for Unmanned Aerial Systems (UAS) pose increasing demands on modeling, planning and simulation capabilities. For example, solar UAS are highly dependent on environmental conditions and must execute autonomous missions lasting several weeks (see Fig. 1).

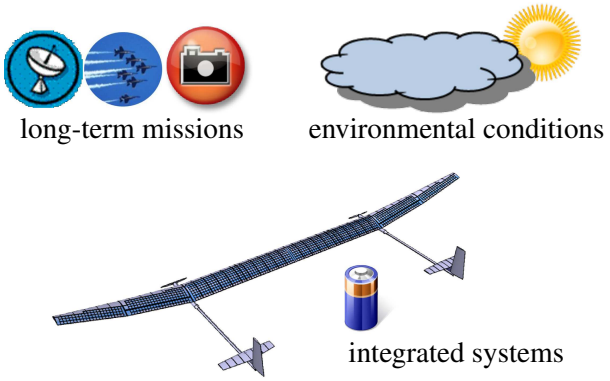


Figure 1: The growing complexity of integrated unmanned aircraft systems, environmental conditions and long-term missions call for efficient and intuitive tools for mission design and simulation.

With growing complexity of the systems, integrated simulation schemes have to be employed, not only modeling the vehicle's flight dynamics but also its avionics systems or even aeroelasticity. Such models have to be as detailed as possible, while maintaining sufficient simulation speed to also run simulations of long-term missions. We have shown previously that Modelica is an excellent tool for this purpose [1].

Additionally, missions of increased complexity and diversity make use of a range of distinctly developed UAS capabilities such as collision avoidance, formation flying or physical interaction with the environment. In order to further improve a system's versatility, developers seek to employ the same system for a variety of purposes, providing as much autonomy to the system as possible (see e.g. [2]).

In order to face these challenges, a flexible, scalable and intuitive scheme for UAS control systems and mission plans has to be provided. Ögren recently proposed behavior trees for this purpose [3]. They are distinguished by their standardized structure providing a mission design scheme, which Champandard argues to combine important advantages of different schemes such as state machines and task planners [4].

However, conventional behavior tree implementations rely on periodically updating the tree's status based on its inputs. This *clocked* or *discrete-time* processing makes previous behavior tree formulations unsuitable for continuous-time simulation of long-term missions. The tree's update rate would have to be chosen high enough to correctly reflect the system's behavior. The smaller integrator time-steps and additional time events would in turn slow down the overall simulation speed considerably.

In order to combine efficient long-term simulations with the capabilities of behavior tree mission plans, a *continuous-time* formulation for behavior trees was thus developed and implemented in Modelica. The present paper describes this formulation and the resulting Modelica library:

- The conventional behavior tree formulation is characterized by discrete-time, sequential, top-down processing of the tree. The new scheme is laid out as continuous-time, event-driven, and bottom-up processing, see Sec. 3. This allows a simulator to chose large integration step-sizes as desired for long-term mission simulation. The formulation can be generalized to other languages supporting event notifications.
- Section 4 introduces the Modelica implementation of the modified system. A library of base tasks with clear internal and external interfaces allows the user to graphically design mission plans and also easily implement new task types. Additional infrastructure is provided to simplify communication with the tree. The processing of an exemplary simple task is shown in Sec. 5.
- The example shown in Sec. 6 show-cases the modular assembly of a mission plan for a solar UAS. The simulation speed of the controlled system is maintained. A comparison to a state-machine based approach using the StateGraph2 library [5] gives identical results.

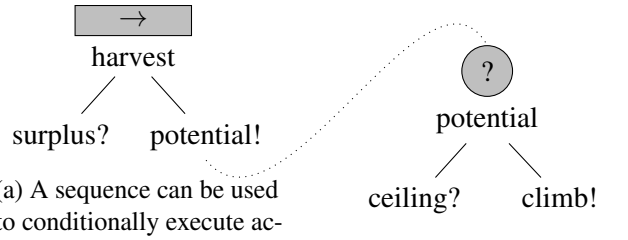
2 Behavior Trees

Behavior Trees are a technique developed for artificial intelligence in computer games. They are first mentioned in this context by Damian Isla [6]. A good introduction is found in Millington's textbook [7]. The approach is introduced to the UAS world by Ögren [3].

Using behavior trees, complex missions are built up using atomic tasks. These can e.g. query the aircraft's state (conditions) or send commands to the underlying control system (actions). A typical combination of the tasks is to conditionally execute an action. A solar UAS e.g. might need to harvest solar energy by maximizing its flight altitude, if a surplus of energy is available. In behavior trees, this is expressed using a sequence of sub-tasks as shown in Fig. 2a.

The task of maximizing potential energy could then be further composed of two alternatives: Either the maximum altitude is already reached or the aircraft has to climb. A set of alternative approaches to a common goal is expressed using a selector as shown in Fig. 2b.

A sequence executes all its sub-tasks in the given order until all sub-tasks are finished successfully. The selector also executes its sub-tasks in the given order, but returns successfully with one successfully finished sub-task. The two basic composite tasks can thus be compared to logical AND and OR operators.



(a) A sequence can be used to conditionally execute actions, e.g. harvesting potential energy based on the condition that a surplus of solar energy is available. Sub-tasks can be decomposed further, such as the "potential" task shown in Fig. 2b.

(b) Selectors are used to dynamically select alternative ways to approach a common goal, such as simply asserting the goal condition of maximum altitude or moving towards it by climbing.

Figure 2: In a behavior tree, simple tasks are connected to a tree in order to achieve more complex goals.

The main advantage of behavior trees for UAS mission management is their standardized and intuitive structure. All atomic tasks describe self-contained and goal-directed behaviors. Higher-level plans are built by intuitively decomposing complex goals into sub-goals. Since all tasks have a common interface and the switching of tasks is determined by implicit logics, the user can modularly interchange arbitrary tasks in the tree. The plans are thus very scalable and human-readable on all levels of the hierarchy. Additionally, behavior trees are inherently memory-free and work with persistent statuses. They thus execute the correct task immediately after a restart or online modification.

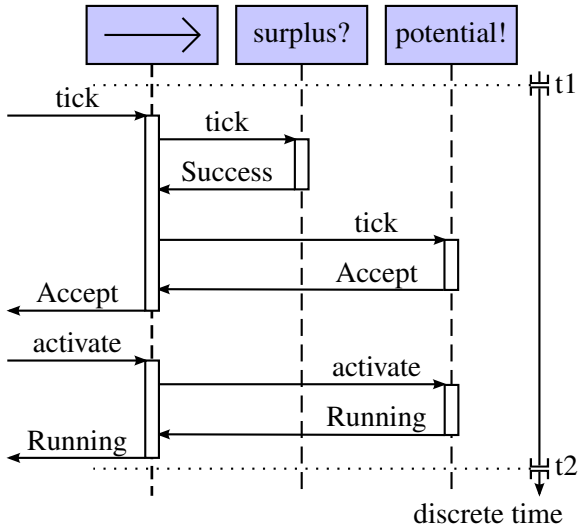
In previous studies, I have augmented behavior trees with state-machine-like entry and exit hooks [8]. The necessary notion of transient tasks is complemented in that paper with an introduction to the processing of basic behavior trees and further discussion of advantages as well as prospects of behavior trees for UAS mission management. A first step towards logical verification and validation of behavior trees is also covered by previous research [9]. Here, the formalism is refined for continuous system simulation.

3 Continuous-time Processing

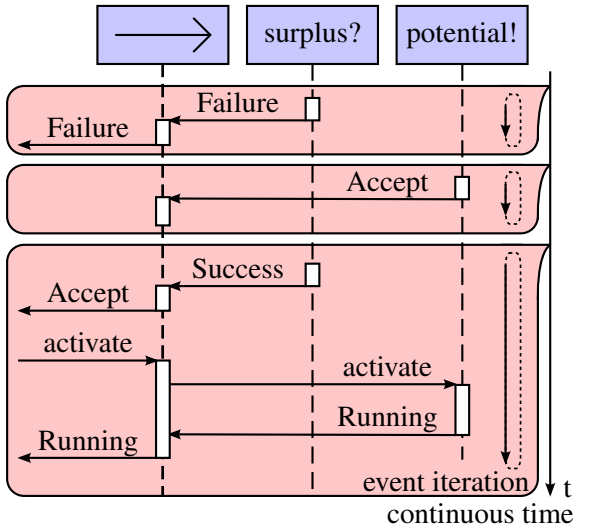
Conventional behavior trees are processed at a fixed update rate. At each instance of time, a tick is issued to the top-most node of the tree. A tick requests a task to report its status back to the superior task. The standard statuses of a task are Success, Failure and Running.

The tick is propagated in a top-down manner through the tree to its leaf nodes. Each composite task, such as a sequence or a selector, sequentially ticks its sub-tasks to determine their statuses. Figure 3a uses a sequence

DOI 10.3384/ECP14096727



(a) In conventional processing, a tick is propagated from the root node of the tree down to the leaf tasks. After all sub-tasks have been ticked, a task returns its own status to its superior task. If necessary, additional activation routines are executed subsequently. This processing is done in each discrete time step.



(b) The proposed continuous-time processing relies on the sub-tasks notifying their superior tasks of status changes. In Modelica, these notifications are generated in event iterations and passed up the tree. Activation procedures are still triggered from the top. This processing can be done in continuous-time.

Figure 3: The processing of tasks in behavior trees is conventionally done sequentially in discrete time. In the new processing scheme, status notifications are passed through the tree in continuous time. Both variants are contrasted here in form of sequence diagrams showing processing of the example sequence from Fig. 2a.

diagram to illustrate the process of ticking the example sequence from Fig. 2a. The sequence is not running at first. In the illustrated time-step, it asserts a surplus of solar energy and then decides to accept being activated. It is subsequently activated by its superior node, and then activates its harvesting sub-task in turn. The separate decision and activation parts of the process are a consequence of the activation/deactivation procedures introduced previously [8].

The downside of this straight-forward processing is that a tick must be issued to the root node with a sufficiently high repetition rate in order to react adequately to changes in the environment. On the Modelica side, this is solved using discrete-time events. These in turn slow down long-term continuous simulations, which cannot make use of large integrator step-sizes anymore and have to handle additional integrator restarts.

In order to overcome this downside, I introduce the bottom-up approach for the decision part of the processing shown in Fig. 3b. Instead of periodically querying the sub-tasks' statuses, the superior task is notified by its sub-tasks about status changes. The superior task may then adapt its own status to the new situation and propagate the new status towards the root of the tree. The activation part of the processing is not changed.

Using this scheme, the example sequence starts with a Failure status, because it cannot assert a surplus of solar energy. The harvesting sub-task can then asynchronously decide to accept being activated. This does not require any status change of the sequence, such that the change event is not passed up the tree. Finally, the surplus sub-task notifies the sequence of its new Success status. The sequence then passes on its Accept status to the higher levels of the tree and is activated as in conventional processing. Each of these status notifications is processed in an event iteration.

If the behavior tree processing is changed in the proposed way, it is relieved of most of the discrete-time events. Instead, state events will be triggered in Modelica whenever a task changes its status. This allows for continuous-time simulation. Additionally, status changes of the behavior tree are resolved instantly as if the tree had an infinitely high update rate. The additional cost of iterating state events is moderate, because typical behavior tree mission plans change their status at a much larger time-scale than the simulation of the controlled system.

The new processing scheme additionally addresses the known limitation of behavior trees in handling events like a finite state machine as pointed out in pre-

vious work [8]. With the presented processing, the foundations are laid to implement event-based behavior trees in any programming language supporting event notifications. Additional work will have to be spent on adding internal or external storage to the tree in order to convert instantaneous events to permanent statuses.

4 Modelica Implementation

In Modelica, all tasks are implemented as models extending from a super-class `Task`, in which the common properties of all tasks are defined. The `Task` model in particular implements the status switching logics of the tasks. The status cycle used in this work is an extended version of the one introduced previously [8]. It makes a distinction between transient tasks and non-transient tasks. Transient tasks such as conditions can be evaluated without activation. Non-transient tasks such as an action must be activated in order to evaluate them.

All possible statuses of a task are defined in a `Status` enumeration. It defines the following eight statuses:

Success and Failure are the standard transient statuses and indicate, if the task (usually a combination of conditions) evaluates successfully or not.

Accept indicates that the task will accept being activated. It marks the entry point to the non-transient part of the status cycle. A task can only enter the non-transient part of the status cycle, if it is activated by its superior task. This status was called Activating in [8].

Activating now designates a task, which was properly activated by its superior task and is currently performing initialization procedures. This status can be used to execute the entry hook of the task. It is introduced here in order to allow for time-consuming initialization procedures. These were not possible previously.

Running denotes a properly initialized, non-transient task during its nominal execution.

Finished and Aborted are the non-transient equivalent statuses to Success and Failure. They indicate, if a task has finished running successfully, or if it failed during its execution. These statuses mark the exit point from the non-transient part of the status cycle. A task can only leave these statuses, if it is deactivated by its superior task.

Deactivating, equivalently to Activating, designates a task, which was deactivated by its superior task and is currently performing finalization procedures. It can be used to run the task's exit hook.

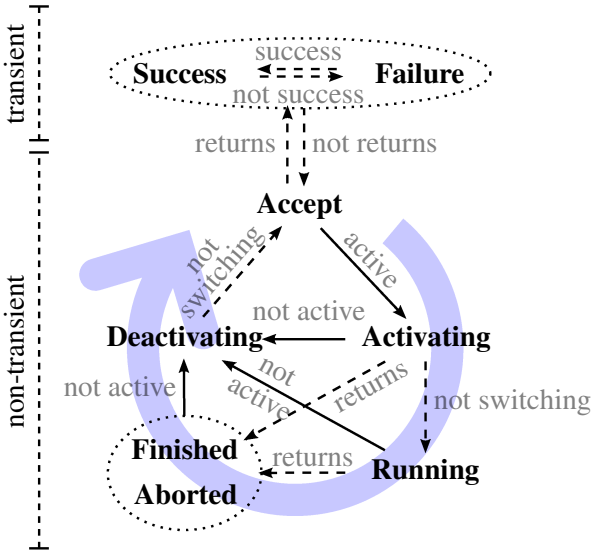


Figure 4: The status of a task is triggered by the internal trigger flags `returns`, `success` and `switching` (- -) or by (de-)activation through the `active` flag (—).

Figure 4 summarizes the complete status cycle. The task may internally determine status changes marked with dashed arrows. The status changes marked with solid arrows can only be initiated by the superior task through activation or deactivation.

Until the task is activated it can choose its status freely from Success, Failure and Accept. This allows transient tasks, especially conditions, to be evaluated without activation. As soon as the task enters the non-transient part of the status cycle by being activated, it is basically bound to iterate its status in the marked clockwise direction. The task can finally leave the non-transient status cycle only if deactivated by its superior task and after passing the Deactivating status.

In order to relieve task designers from coping with the correct implementation of the status cycle, three additional trigger flags are introduced. These indirectly trigger a task's status:

returns determines, if the task can be evaluated in the sense of returning a successful or unsuccessful status or if the task needs to be activated first.

success determines the correct status, if the task has a returning status.

switching is used to mark initialization and finalization of tasks. It can be used e.g. while powering up resources needed by the task.

The task's status can now be uniquely determined using these trigger flags and an additional `active` flag provided by the superior task. The trigger flags are also illustrated in Fig. 4 by the labels of the arrows.

DOI 10.3384/ECP14096727

The appropriate equation is common to all tasks and a base `Task` model is thus provided (see Listing 1). It contains the status switch logics, the three internal trigger flags `returns`, `success` and `switching` as well as a public `uplink` connector to the superior task. In this way, the status dynamics of all tasks are forced to be consistent and new task types can be conceived by simply assigning values to the trigger flags.

Listing 1: The basic `Task` interface contains an `uplink` to the superior task, three additional status triggers and the status switch logics. The function `f()` implements the status cycle shown in Fig. 4.

```
partial model Task
  UpLink  uplink;
protected
  Status  status = uplink.status;
  Boolean active = uplink.active;
  Boolean returns;
  Boolean success;
  Boolean switching;
equation
  status = f(pre(status),
             active,
             returns,success,switching);
end Task;
```

A `Condition` is e.g. defined by Listing 2. It can always be evaluated to Success or Failure and thus fixes the `returns` flag to `true`. The `success` flag determines whether to return Success or Failure. It can be filled by arbitrary conditional equations, e.g. using an additional `BooleanInput`. Because a condition cannot be activated, the `switching` flag must actually only be filled in order to balance the model.

Listing 2: Defining a new `Condition` task is done by binding the trigger flags to boolean expressions.

```
model Condition
  extends Task;
  BooleanInput u    "A boolean input";
equation
  success  = u;    //is used as condition
  returns  = true; //and always returned
  switching= false;// => Not used!
end Condition;
```

All inter-task communication described in Fig. 3 is handled by the `Task`'s `UpLink` connector and corresponding `DownLink` connectors of the superior tasks. The `UpLink` connector is outlined in Listing 3. The `status` variable carries the status information, while the `active` flag is used by the composite tasks to activate or deactivate their sub-tasks. Using these connectors as public interfaces, single tasks can be connected in a tree to almost arbitrarily complex mission plans.

Listing 3: The `BehaviorTrees` `UpLink` connector for connecting sub-tasks to their respective superior task passes the sub-task's `status` up the tree and receives an `active` flag from the superior task.

```
connector UpLink
  output Status status "The task status";
  input Boolean active "(De-)Activation";
end UpLink;
```

Using the interfaces described above, a library of basic tasks is provided. The library's structure is shown in Fig. 5. It includes the most common tasks encountered in behavior trees such as `Condition`, `Action`, `Selector` and `Sequence`. The `Root` task is needed to steer the overall execution of the tree.

Additionally, a communication structure is provided in order to simplify the information flow between the tree and its environment (`Blackboard`). The system relies on a global memory block using Modelica's `inner/outer` functionality. Continuous-time `Real` variables can be written to the system using the `SetReal` block. Each slot is provided with a name and an active flag. The `GetReal` block retrieves the currently active value with a given name from the blackboard. Using this structure, it is possible for several tasks to write to the same input of the controlled system. It especially allows to also encapsulate continuous-time controllers in tasks and to switch between them automatically based on the current behavior tree status.

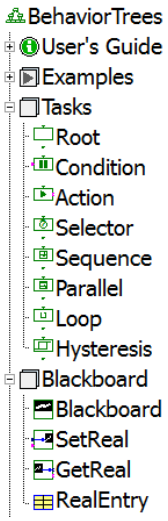


Figure 5: The `BehaviorTrees` library contains a number of standard task implementations. It also provides infrastructure to facilitate communication with the tree.

5 Processing of a Sequence

In this section, the processing of the last event shown in Fig. 3b is detailed with respect to the implementation described in the previous section. The `Sequence` task uses the implementation shown as pseudo-code in Listing 4. It first processes the sub-tasks' statuses to determine its own status triggers. These can then be processed by the standardized status cycle implementation as shown in Fig. 4. If the sequence is active, it passes on this flag to its first accepting sub-task.

Listing 4: The Modelica pseudo-code for the implementation of a `Sequence Task` determines the status triggers and steers the active flag of its sub-tasks.

```
model Sequence
  extends Composite "Task with sub-tasks";
equation
  returns   = not any substatus[:] is
              Accept, Activating, Running;
  success   = all substatus[:] are
              Success or Finished;
  switching = any substatus[:] is
              Activating, Deactivating;
  // <-- Apply status cycle from Task
  if active then
    subactive[first accepting] = true;
  end if;
end Sequence;
```

Initially, the `surplus` sub-task has the status `Failure` and the `potential` sub-task has the status `Accept`. The sequence consistently returns `Failure`.

When the `surplus` status changes to `Success`, the sequence's `returns` flag changes to `false`. Applying the status cycle, the sequence changes its status to `Accept` and passes on this status to its superior task.

According to the switch logics contained in the superior tasks, the sequence's `active` trigger is then changed to `true`. The sequence passes on this value to its first accepting sub-task, the `potential` energy harvesting. This action then changes its status to `Activating`, which in turn changes the sequence's `switching` trigger to `true`. According to the status cycle, the sequence also assumes `Activating` status and passes it up the tree.

Eventually, the `potential` sub-task generates a new event, when it switches its status to `Running`. The sequence adapts to this change by changing its own `switching` flag to `false`, applying the status cycle, and passing its new `Running` status to its superior task.

Figure 6 shows a sequence diagram of this sequence of events. The single final event from Fig. 3b is split in three events here. Depending on the superior tasks and the sub-tasks, these events can also happen immediately one after the other. An action without activation procedure e.g. immediately leaves the Activating status. This combines the latter two events into one.

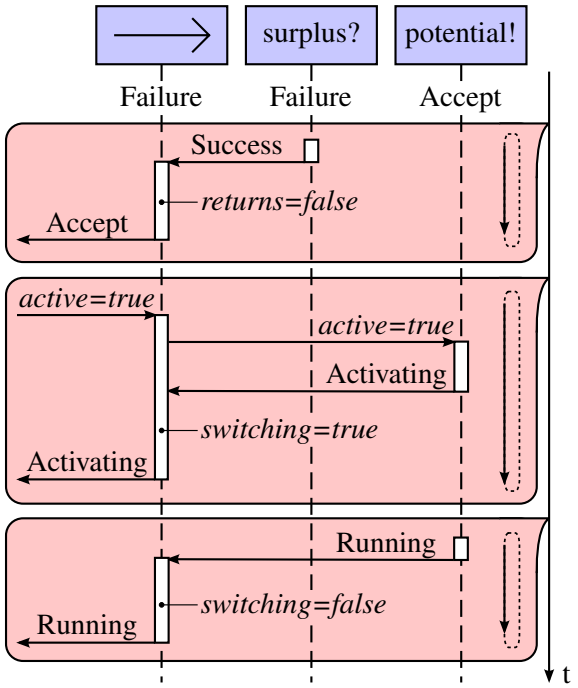


Figure 6: The detailed processing of a sequence activating its potential sub-task includes the status triggers.

6 Application Example

The described system is used for show-casing its applicability to actual UAS mission scenarios. To this end, a mission plan is built for an electric high-altitude solar-powered aircraft. This plan is used to steer the integrated simulation model developed in previous work [1]. The aircraft is implemented as the non-linear inverse of a point-mass model of the flight dynamics with two controlled propellers. It has 26 individually controlled solar panels. The total number of continuous-time states for the aircraft is 72.

The plan is divided in a longitudinal and a lateral part. The lateral part steers the aircraft towards a holding position and commands a holding pattern at this point. This part is not described here in detail. The longitudinal part of the plan is shown in Fig. 7. It consists of two sub-goals. The first is to ensure that a maximum of solar energy is stored in the aircraft. The second includes the main mission, such as operating a communication relay at the holding position. In this example, it is simplified by issuing an altitude holding command.

The energy-maximizing plan is comprised of two alternative plans. One is used to harvest solar energy. It

DOI 10.3384/ECP14096727

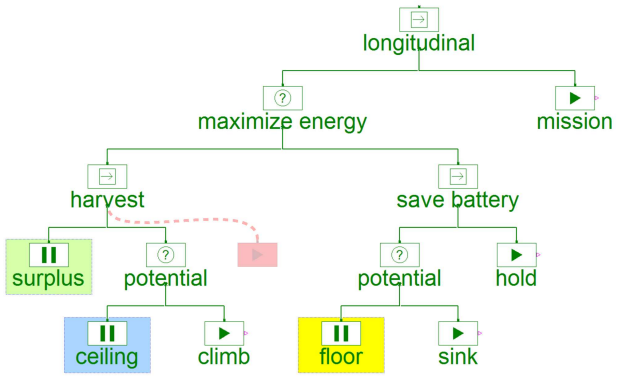


Figure 7: The `BehaviorTrees` mission plan uses encapsulated inputs and outputs specialized for the specific UAS application. The conditions are marked with colored frames for comparison with the transitions in Fig. 8. The plan's modularity is shown by introducing a new sub-task of the harvest task (shaded, dashed line). Only one additional connection is required for this extension.

is described already in Sec. 2 as the introductory example. It commands the aircraft to climb to its maximum altitude, whenever a surplus of solar energy is available. The second alternative ensures that a minimum of energy is used, if no energy can be stored. To this end, the aircraft first descends to a minimum altitude and holds this altitude in a way, which minimizes battery use. This type of using altitude to store potential energy is called a "jojo" mission. It is important to notice, that in this plan formulation, the main mission is only allowed to be executed, if no energy task of higher priority needs to be executed.

Figure 8 shows a `StateGraph2` [5] implementation equivalent to the behavior tree. It consists of the four states `mission`, `climb`, `sink` and `hold` as well as numerous transitions with conditions equivalent to the behavior tree logic. The corresponding state graph transitions and behavior tree conditions are marked with colored frames in Fig. 7 and 8 respectively. It can be noted, that some transitions have to be reused multiple times in order to replicate the behavior tree's results.

The advanced modularity of the behavior tree can be appreciated even more, when an additional task is introduced. Figures 7 and 8 also include a sketch of loading a fuel cell as an additional energy harvesting task. While the behavior tree only requires one additional connection, the state graph has to be extended with a number of transitions. The extension is done here in an ad-hoc way and optimizations of the state machine are possible. However, the typical mission designer will appreciate the facilities to create ad-hoc plan changes in an efficient way such as provided by the `BehaviorTrees` library.

The results of both the `BehaviorTrees` and the `StateGraph2` mission plans are shown in Fig. 9. A full day of flight (86 400 s) is shown including a full jojo mission between 6000 m and 13 000 m of altitude. The results of both mission plans are identical: The simulation starts before sunrise, such that the aircraft holds its lower altitude limit. When the sun rises at about 07:50, the batteries start charging. Only when the batteries are fully charged at about 11:30, the aircraft climbs to its mission altitude. With decreasing solar energy in the early evening, the aircraft starts to sink to its lower altitude limit again.

Figure 9 also shows the related current productions. The repeated switching with critical battery stems from highly different power demands for the altitude holding and sinking tasks in combination with discontinuous commands from the mission plan. Providing fully continuous switching will alleviate this behavior.

In order to compare the computational complexity of the different implementations, the required CPU time on a standard laptop computer and the generated number of time- and state-events are given in Tab. 1. The values are compared for the discrete-time formulation of the behavior tree and the continuous-time `BehaviorTrees` modification. The tree is ticked once per minute in the discrete-time case. The important reduction of both time events and execution time can be seen. The continuous-time formulation saves about 80 % of the simulation time and removes all time events. For completeness, the execution times of the `StateGraph2` implementation are also given, as well as the results from simulating only the aircraft model with the recorded command inputs for the same period.

Table 1: The execution measures of the different configurations are determined for a single simulation experiment of the discrete-time and the continuous-time behavior tree as well as the state graph implementation and a direct simulation with recorded inputs. CPU time, number of time events and number of state events are extracted from the simulation log file.

Configuration	CPU time	time- / state events	
Discrete	368 s	1442	249
Continuous	72 s	0	247
State graph	76 s	60	242
Direct inputs	66 s	25	183

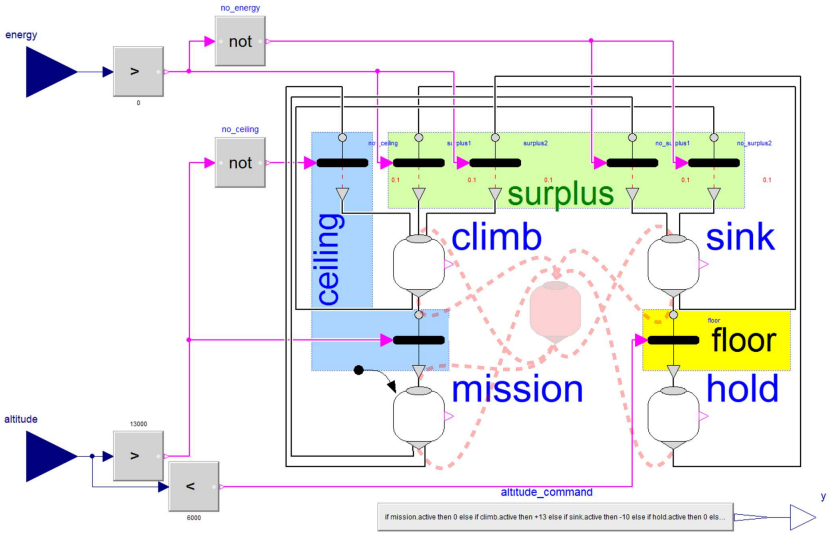
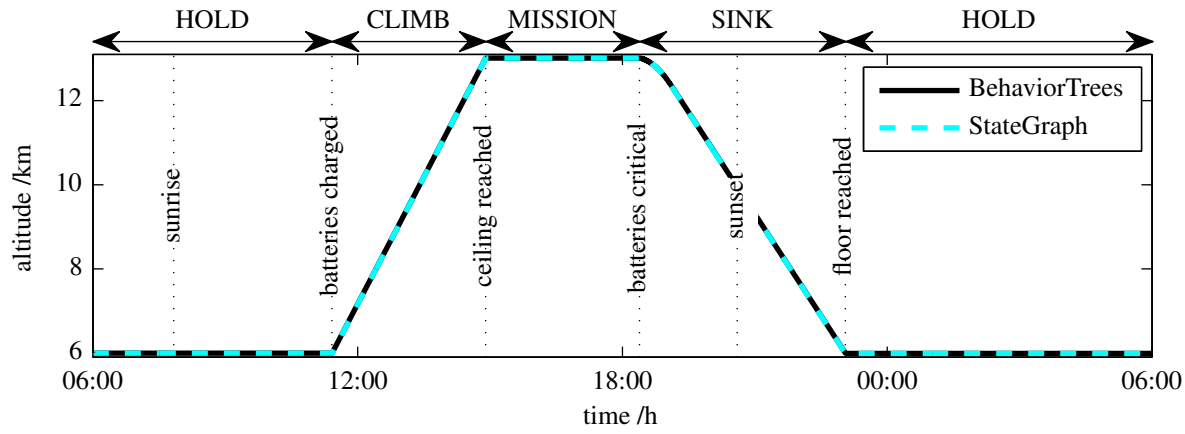
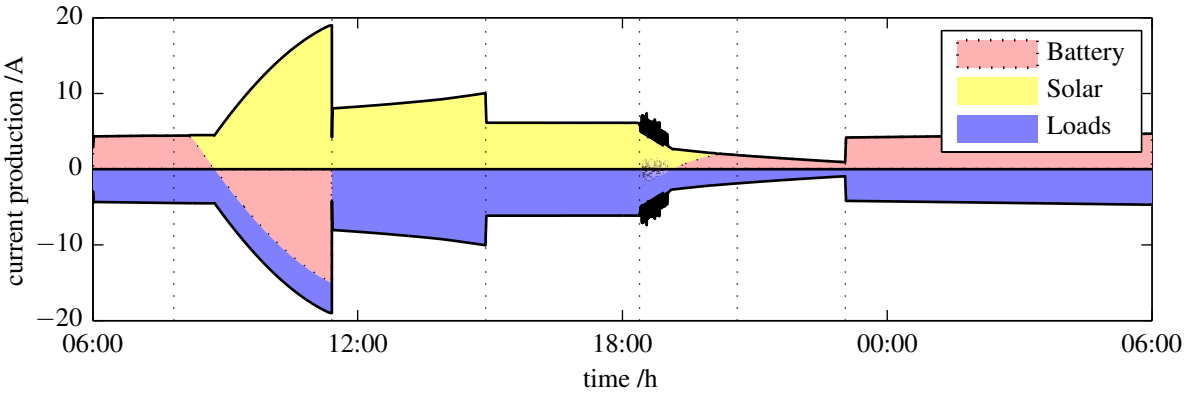


Figure 8: The `StateGraph2` mission plan consists of four places corresponding to the actions of the `BehaviorTrees` mission plan. The conditions of the behavior tree are translated into transitions. They are marked with colored frames for comparison with Fig. 7. It can be seen that the state machine implementation requires a number of redundant transitions. The complexity of this implementation becomes more visible, when an additional task is introduced similarly to the one included in Fig. 7 (shaded, dashed lines).



(a) The altitude results for a full day of flight are identical for both mission plans.



(b) Also, the energy production and consumption patterns are equal for both simulations.

Figure 9: Both mission plans are simulated with an integrated aircraft model for a full day of flight. The results are identical for both plans.

DOI 10.3384/ECP14096727

7 Conclusions and Outlook

The conventional discrete-time formulation of behavior trees can be argued unsuitable for long-term UAS mission simulation in Modelica because of its forced update rate. A continuous-time formulation is thus devised and implemented in the Modelica `BehaviorTrees` library. The formulation allows the simulator to choose large step-sizes, allowing for long-term mission simulation with behavior tree mission plans in Modelica. The approach relies on passing event notifications between composite tasks and their sub-tasks.

The new formulation also addresses the limitations of previous behavior tree formulations with respect to event-handling. It provides the capability needed to process behavior trees based on events also in languages other than Modelica. By providing a suitable data storage, the approach can be used to create event-driven behavior trees also in real-time environments such as flight computers.

The Modelica library described in this paper presents clear public and internal interfaces, allowing the user to design mission plans graphically on the one hand and to implement new tasks types on the text layer on the other hand. The behavior tree infrastructure is complemented with a communication infrastructure allowing to conveniently pass signals from and to the plan without direct connections.

A plan implemented with the `BehaviorTrees` library gives identical results compared to an equivalent plan using the `StateGraph2` formalism. The simulation speeds confirm that long-term simulations are possible with both methods. However, the plan's modularity is increased using the `BehaviorTrees` facilities as compared to a `StateGraph2` implementation.

The comparison of results and timings should be regarded as an initial qualitative result. It indicates that behavior trees can indeed conveniently be used to steer a solar-powered high-altitude aircraft on long-term missions. Quantitative evaluations and comparisons to conventional state machines and discrete real-time implementations still need to be carried out. Formal validation and verification need to be addressed in order to guarantee consistent behavior between the continuous and discrete formulations.

The `BehaviorTrees` library is currently only used in internal studies and research projects. A commercial, or open-source, release is currently not planned but not ruled out either. The library will be further developed in the scope of a doctoral project and interested users are invited to contact the author for further information.

Future versions of the library will make use of the new synchronous elements provided by the Modelica 3.3 specification. These should provide for even better performance and robustness. Comparisons to pure Modelica 3.3 state charts should give similar results as the comparison to `StateGraph2`: Similar performance with improved modularity. Additional improvements can be made with respect to repeated switching by providing for continuous switching, where possible.

In summary, the prospects of using `BehaviorTrees` mission plans in Modelica are excellent, both in usability and in performance. The approach can be valuable not only for UAS applications, but also for other fields such as vehicle test automation. Future research effort will additionally be spent on deploying such mission plans to actual flight computers using automatic code generation and the functional mockup interface.

References

[1] 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, *Advances in Aerospace Guidance Navigation and Control – Selected Papers of the Second CEAS Specialist Conference on Guidance, Navigation and Control*, pages 535–548. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-38252-9.

[2] Maximilian Laiacker, Andreas Klöckner, Konstantin Kondak, Marc Schwarzbach, Gertjan Looye, Dominik Sommer, and Ingo Kossyk. Modular scalable system for operation and testing of UAVs. In *American Control Conference*, pages 1462–1467, Washington, DC, 17-19 June 2013. IEEE. ISBN: 978-1-4799-0176-0.

[3] 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.

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

[5] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven Erik Mattson, and Charlotta Johnsson. A new formalism for modeling of reactive and hybrid systems. In *Proceedings of the 7th International Modelica Conference*, pages 364–377. Linköping University Electronic Press, 2009.

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

[7] Ian Millington and John Funge. *Artificial intelligence for games*. Morgan Kaufmann, Burlington, MA, 2nd edition, 2009. ISBN: 978-0123747310.

[8] Andreas Klöckner. Behavior Trees for UAV Mission Management. In Matthias Horbach, editor, *INFORMATIK 2013: Informatik angepasst an Mensch, Organisation und Umwelt*, volume P-220 of *GI-Edition-Lecture Notes in Informatics (LNI) - Proceedings*, pages 57–68, Koblenz, Germany, 16-20 September 2013. Gesellschaft für Informatik e.V. (GI), Köllen Druck + Verlag GmbH, Bonn. ISBN 978-3-88579-614-5.

[9] Andreas Klöckner. Interfacing Behavior Trees with the World Using Description Logic. In *AIAA Guidance, Navigation, and Control Conference*, Boston, MA, 19-22 August 2013. AIAA. AIAA 2013-4636. ISBN 978-1-62410-224-0.

DOI 10.3384/ECP14096727