Trapping Behavior Trees in Esterel

Alexander Schulz-Rosengarten*, Michael Mendler†, Joaquin Aguado†, Malte Clement* and Reinhard von Hanxleden*
*Kiel University and †Bamberg University

***Abstract*—Behaviour Trees (BTs) are a formalism for specifying behavior in a modular way. Originally from gaming applications, they have recently gained attention for industrial automation and robotics control as well. However, so far there has been little work in terms of formalization or grounding in other established programming formalisms. We propose some syntactic sugar for Esterel to capture directly the basic mechanisms of BTs. This grounds the essence of BTs in a well-established synchronous programming language.**

I. Introduction

Behaviour Trees (BTs) originated in the gaming industry but recently gained popularity for real-world applications, such as robotics [1]. They use a model-based approach with a simple and intuitive tree structure and a lean set of flow elements to control the execution of tasks, as detailed further in Sec. II. As it turns out, BTs have much in common with synchronous programming [2]. They both follow a tick-based execution regime, target reactive systems, favor modularity, and feature modeling concurrent behavior. Yet, in the literature, BTs are usually discussed in relation to Finite State Machines (FSMs), Teleo-reactive Programs, or Decision Trees [1]. Synchronous language have a lot to offer and can aid in hardening BTs for the domain of embedded and safety-critical systems. They provide well-formed semantics with deterministic concurrency models, facilitate verification, and come with rich ecosystem for programming, causality analyses, compilation, and modularization. Here, we present a first look at how BTs and synchronous programming can be brought together, using Esterel as an example. Future work will investigate the benefits the synchronous concepts might yield in the BT context.

II. What are Behavior Trees?

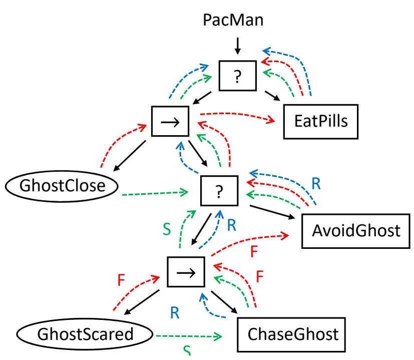

Fig. 1. Pac-Man behavior tree (downward solid lines) with its associated decision graph (upwards dashed lines) routing of Failure (F), Success (S) and Running (R).

The root of a BT starts by generating *ticks* (tokens) that travel along the tree down to the *execution nodes* (leaves). Execution nodes include *actions* (depicted as rectangles) and *conditions* (depicted as ovals). For the Pac-Man in Fig. 1 (an example taken from [1]), the actions are ChaseGhost, AvoidGhost and EatPills, and the conditions are GhostClose and GhostScared. A node is activated once it gets the tick from its parent, then it executes and responds back to its parent. The response of an action can be either Running (not terminated yet), Success (goal achieved) or Failure (finished unsuccessfully). The response of a condition can only be Success (true) or Failure (false). The BT control flow is encoded using the internal (non-leaves) *control flow* nodes: *sequence* and *fallback*. There is also a *parallel* control flow node for BTs but this is not employed in the present paper. A *sequence* node (a box with a right-pointing arrow →) sends ticks to its children from left to right. If a child returns Success, the next child is ticked; if there are no further children, then the sequence returns Success as well. If, however, a child returns another status (Running or Failure), the sequence returns that status as well and no further child is ticked. For the BT in Fig. 1, a sequence node allows the Pac-Man to execute action ChaseGhost whenever the condition GhostScared holds. Note that in Fig. 1, the Success (Failure, Running) flow is indicated by the green (red, blue) dotted arrows. A *fallback* node (a box with question mark ?) also sends ticks to its children from left to right. It returns Failure when all children return Failure. Otherwise, it returns Running or Success as soon as one of its children does so. The BT of Fig. 1 includes a fallback at the root which tells the Pac-Man to switch between action EatPills and its left subtree. There is another fallback that ensures that AvoidGhost executes when GhostClose is true. In this form, the Pac-Man avoids or chases ghosts (AvoidGhost or ChaseGhost) depending on whether the ghosts are close or scared (GhostClose and GhostScared). Thus, the Pac-Man eats pills (EatPills) when it is not avoiding or chasing ghosts.

III. Esterel Programming of BT (BTEsterel)

As a reactive orchestration language for concurrent memory actions, BTs bears striking similarities with Esterel [3]. Like in BTs, Esterel is heavily based on traps. In addition, Esterel is a fully-fledged programming language that offers many features not present in core BTs. It provides a rich set control-flow constructs, statically typed data and thread-local memory to program complex temporal behaviour and reactive control algorithms. It permits modules to communicate with each other through *valued signals* and still guarantees deterministic execution, bounded memory and deadlock-freedom, even in the presence of run-time concurrency. BTs can be seen as a domain-specific syntactic extension of Esterel. BTs are mapped to Esterel modules that communicate via signals. Every tick of the BT corresponds to one macro-step of the module and—for memory-free BTs—the Esterel code executed at each tick is always the same. Inside a module, the control flow of BT nodes is implemented with Esterel's trap mechanism.

A representation of the Pac-Man example is given in Fig. 2a. The signals GhostClose and GhostScared implement the BT condition nodes. The signals ChaseGhost, AvoidGhost and EatPills are used to implement the BT actions. The completion status of an input is tested by present c then P else Q end. If c completes by SUCCESS the signal is present and P is executed. If c completes by FAILURE then its signal status is absent and Q is executed, instead. For instance, in lines 8–10 of Fig. 2a the presence of GhostClose leads to the execution of the Esterel trap exit _btsucc and its absence raises the trap exit _btfail. The sequence and fallback operators, which are abbreviated

```
btfallback t1 btfb t2 btfb end btfallback
btsequence t3 btseq t4 btseq end btsequence
```

are implemented using Esterel's generic trap handler [3] trap t in P do Q end for *trap identifier* $t \in \{\_btsucc, \_btfail\}$. It names the lexical scope for all exceptions exit t inside a task P that will trigger the instantaneous preemption and exit from the trap's scope with immediate continuation in task Q. When P never terminates, which we assume, this is the same as trap t in P end; Q where ; is Esterel's sequential composition operator. Hence, the syntactic abbreviations btfallback and btsequence above get expanded into

```
trap _btfail in t1 end trap; t2
trap _btsucc in t3 end trap; t4,
```

respectively, as is seen in Fig. 2b. Here, we assume that BT actions are triggered by emitting output signals. So, if the action ChaseGhost completes with RUNNING every time it is ticked, we have the coding seen in lines 20-21 of Fig. 2a:

```
emit ChaseGhost; exit _btrun
```

The _btrun trap is handled by the wrapper construct behaviortree P end behaviortree. Here, this outer-most wrapper makes the BT complete the tick by by emitting an extra output signal BehaviortreeRunning (line 34), pausing (line 35) and a loop to repeat the full code for the next tick (lines 6 and 36).

IV. Summary & Conclusion

We propose a concept for representing BTs in Esterel using traps. It illustrates the common ground between BTs and synchronous languages and acts as a proof of concept showing that reactive BTs can be directly embedded in the Esterel programming environment. Yet, this is only a first step toward a more comprehensive combination of BTs and synchronous languages. We plan to extend our concept into a more fully elaborated language extension, including grounding BTs in

```
1  module PacMan
2  input GhostClose,
        GhostScared;
3  output ChaseGhost,
        AvoidGhost, EatPills;
4
5  behaviortree
6   btfallback
7    btsequence
8     present GhostClose
9      then exit _btsucc
10     else exit _btfail
11    end;
12   btseq
13    btfallback
14     btsequence
15      present GhostScared
16       then exit _btsucc
17       else exit _btfail
18      end;
19     btseq
20      emit ChaseGhost;
21      exit _btrun
22     end btsequence
23    btfb
24     emit AvoidGhost;
25     exit _btrun
26    end btfallback
27   end btsequence
28  btfb
29   emit EatPills;
30   exit _btrun
31  end btfallback
32 end behaviortree
```

(a) With BT syntax extensions, before expansion.

```
1  module PacMan
2  input GhostClose, GhostScared;
3  output ChaseGhost, AvoidGhost,
        EatPills;
4  output BehaviortreeRunning;
5
6  loop
7   trap _btrun in
8
9    // Application logic
10   trap _btfail in
11    trap _btsucc in
12     present GhostClose
13      then exit _btsucc
14      else exit _btfail
15     end;
16    end trap;
17    trap _btfail in
18     trap _btsucc in
19      present GhostScared
20       then exit _btsucc
21       else exit _btfail
22      end;
23     end trap;
24     emit ChaseGhost;
25     exit _btrun
26    end trap;
27    emit AvoidGhost;
28    exit _btrun
29   end trap;
30   emit AvoidGhost;
31   // End of application logic
32
33  end trap;
34  emit BehaviortreeRunning;
35  pause
36 end loop
```

(b) Plain Esterel, after expanding BT extensions.

Fig. 2. Pac-Man as an Esterel Module, with and without BT syntax extensions.

the formal semantics of Esterel. We will further refine the modularity of our solution, enabling instantiating BTs as sub-trees. Our solution with traps and their exit codes corresponds k-BTs [4] and is, as far as we are aware, the first proposal for constructing a full-blown programming language for k-BTs. Parallel nodes often pose a challenge [5] in the BT semantics. Synchronous languages have a lot to offer here. Providing deterministic concurrency is one of their core capabilities, which would directly benefit programming with BTs.

References

[1] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages Twelve Years Later," in *Proc. IEEE, Special Issue on Embedded Systems*, vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[3] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[4] O. Biggar, M. Zamani, and I. Shames, "A principled analysis of Behavior Trees and their generalisations," *CoRR*, vol. abs/2008.11906, 2020. [Online]. Available: https://arxiv.org/abs/2008.11906

[5] M. Colledanchise and L. Natale, "Handling concurrency in behavior trees," *IEEE Transactions on Robotics*, vol. 38, no. 4, pp. 2557–2576, 2022.