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. 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.

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).

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 \( \text{exit}_\text{btsucc} \) and its absence raises the trap \( \text{exit}_\text{btfail} \). The sequence and fallback operators are abbreviated

\[
\text{btfallback} \quad \text{btsequence} \quad \text{btseq} \quad \text{btsequence} \\
\text{btb} \quad \text{btseq} \quad \text{btseq} \quad \text{btsequence}
\]

are implemented using Esterel’s generic trap handler \( \text{trap} t \in P \) do \( Q \) end for \( \text{trap identifier} t \in \{ \text{btsucc}, \text{btfail} \} \). It names the lexical scope for all exceptions \( t \) 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 \( \text{trap} t \in P \) end; \( Q \) where \( ; \) is Esterel’s sequential composition operator. Hence, the syntactic abbreviations \( \text{btfallback} \) and \( \text{btsequence} \) above get expanded into

\[
\text{trap}_\text{btfail} \quad \text{end trap}_\text{btfail} \\
\text{trap}_\text{btsucc} \quad \text{end trap}_\text{btsucc}
\]

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 \( \text{RUNNING} \) every time it is ticked, we have the coding seen in lines 20–21 of Fig. 2a

\[
\text{emit}_\text{ChaseGhost}; \text{exit}_\text{btrun}
\]

The \( \text{btrun} \) trap is handled by the wrapper construct \( \text{behaviortree} P \) end \( \text{behaviortree} \). Here, this outer-most wrapper makes the BT complete the tick by by emitting an extra output signal \( \text{BehaviortreeRunning} \) (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 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