Correct-by-construction execution and compilation of Behavior Trees

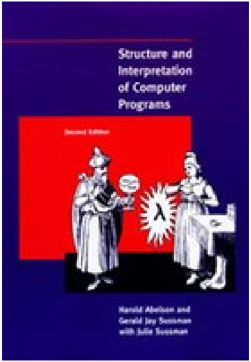
Alberto Tacchella

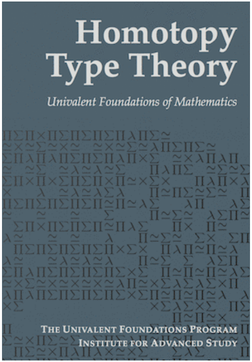
FBK, Trento

May 3, 2019

The long and winding road

My background: mathematical physics, geometry





. . . type theory, logic, functional programming

The CARVE project

- Collaboration between IIT, Unige, UTRC
- Part of the **RobMoSys** EU project

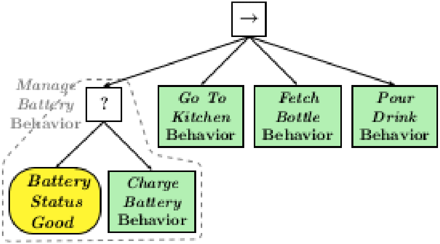
Main goals:

- to introduce a formalism for modeling composable and reusable *robotic behaviors* (= routines);
- to develop a set of verification tools for increasing confidence in the *correct execution* of those behaviors.

Key tool: Behavior Trees (BTs) as a hierarchical abstraction for modeling component execution.

Behavior Trees

BTs are a graphical model that can be used to specify how an autonomous agent switches between different tasks.



Compared to FSMs, they emphasize:

- *modularity* (intrinsically recursive),
- *reactiveness* (no closed-world assumptions).

Problem: no formal semantics for them.

The Coq theorem prover

- Developed at INRIA, France starting in 1984 (Gérard Huet, Thierry Coquand)
- Officially billed as an *interactive theorem prover* or *proof assistant*
- Some limited forms of automation (more can be implemented using *tactics*)
- Formally based on a version of Martin-Löf's dependent type theory called the *calculus of inductive constructions* or CoC
- Via the Curry-Howard correspondence, CoC terms can be translated into purely functional programs. This is exploited by Coq's program extraction mechanism, which targets "regular" programming languages such as OCaml, Scheme and Haskell.

Semantics for BTs via shallow embedding into CoC

Our idea: specify an operational semantics for Behavior Trees by embedding the “language of BTs” into CoC.

Detailed plan:

- define a data type of behavior trees, parametric on a set of basic skills;
- write a function `tick` realizing the (informally specified) operational semantics of BTs:
- use program extraction to get a working interpreter for BTs.

Optionally: use Coq to *prove properties* about the generated interpreter (e.g. always terminates).

BT embedding, binary implementation

```
Inductive nodeKind: Set :=
  Sequence | Fallback | Parallel1 | Parallel2.

Inductive decKind: Set :=
  Not | IsRunning.

Inductive btree: Set :=
| Skill: skillSet -> btree
| Node: nodeKind -> string -> btree -> btree -> btree
| Dec: decKind -> string -> btree -> btree.

Inductive return_enum := Runn | Fail | Succ.

Definition skills_input := skillSet -> return_enum.
```

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
  | Skill s => input_f s
  | Node k _ t1 t2 =>
    match k with
    | Sequence =>
      match (tick t1 input_f) with
      | Runn => Runn
      | Fail => Fail
      | Succ => (tick t2 input_f)
      end
    | Fallback =>
      match (tick t1 input_f) with
      | Runn => Runn
      | Fail => (tick t2 input_f)
      | Succ => Succ
      end
    end
  | Dec k _ t =>
(* ... *)
  end.
```

BT embedding, arbitrary-branching implementation

```
Inductive btree: Set :=
| Skill: skillSet -> btree
| Node: nodeKind -> string -> btforest -> btree
| Dec: decKind -> string -> btree -> btree
with btforest: Set :=
| Child: btree -> btforest
| Add: btree -> btforest -> btforest.
```

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
  | Skill s => input_f s
  | Node k _ f =>
    match k with
    | Sequence => tick_sequence f input_f
    | Fallback => tick_fallback f input_f
(* ... *)
  end
with tick_sequence (f: btforest) (input_f: skills_input) :=
       match f with
       | Child t => tick t input_f
       | Add t1 rest => match tick t1 input_f with
                        | Runn => Runn
                        | Fail => Fail
                        | Succ => tick_sequence rest input_f
                        end
       end
(* ... *)
```

Program extraction to OCaml

```
let rec tick t input_f =
  match t with
  | Skill s -> input_f s
  | Node (k, _, f) ->
    (match k with
     | Sequence -> tick_sequence f input_f
     | Fallback -> tick_fallback f input_f
(* ... *)

and tick_sequence f input_f =
  match f with
  | Child t -> tick t input_f
  | Add (t1, rest) ->
    (match tick t1 input_f with
     | Succ -> tick_sequence rest input_f
     | x -> x)
```

$\sim$ 350 lines of generated ($=$ trusted) ML code for the final version (vs. $\sim$ 200 lines written by hand – the vast majority for opening & parsing the XML input file)

Alternative semantics in terms of FSMs

UTRC developed another semantics for BTs by defining a translation to Hierarchical Finite State Machines (see CARVE deliverable D3.2).
This is also useful for verification purposes, both offline (model checking) and online (monitoring).

- How to reconcile the UTRC semantics with the one used for the interpeter?

Model-checking toolchain chosen: NuSMV+OCRA.

Problem: no way to interface directly Coq with NuSMV. In order to be able to formally manipulate SMV specifications, we decided to embed into Coq a restricted subset of the SMV language.

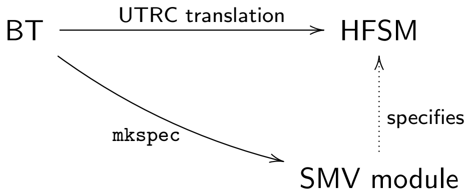
MicroSMV

Highly simplified version of SMV featuring only two basic types (booleans and symbolic enums), a reduced set of expressions, DEFINE macros, ASSIGN constraints and parametric modules.

```
(AddA (init (Mod "top_level_bt" (Id "enable"))
            (BConst smvT))
      (LastA (next (Mod "top_level_bt" (Id "enable"))
                   (Neg (Equal (Qual (Mod "top_level_bt"
                                          (Id "output")))
                               (SConst "none")))))).
```

```
"init(top_level_bt.enable) := TRUE;
next(top_level_bt.enable) := !(top_level_bt.output = none);
"
     : string
```

Specification extractor



We implemented an automated tool (`mkspec`) to translate a BT into a MicroSMV module specifying the corresponding HFSM, as described in the UTRC semantics (see the D5.1 document, Section 3 for the details).

This tool is also (mainly) obtained by program extraction starting from Coq sources ($\sim$ 1400 lines of generated code).

Relationship between the two semantics

Ideally, one would like to prove that the two semantics are equivalent. A possible way do to this is:

- define inside Coq a notion of execution for HFSMs specified in the MicroSMV language, and
- prove that for any term t: btree and for any input i one has exec (translate t) i = tick t i.

Practical problem: the needed proof is nontrivial and requires a good amount of Coq expertise & automation.
Principle problem: even if we had such a proof, the actual model checking step is performed by NuSMV, whose code is *totally unrelated* to the above-defined notion of execution.

The only way to reconcile the two semantics in a fully formal way would be to develop a (formally verified) model checker in Coq and use *that* model checker to perform the validation step. (Of course, this route has problems of its own.)