CARVE: Composable Robot Behavior with Verification

Model transformation and correctness proof

The CARVE project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732410, in the form of financial support to third parties of the RobMoSys project.

Leader for the deliverable:	Università degli studi di Genova (UNIGE)
Document type:	Report
Dissemination level:	Public
Revision:	1.0
Delivery date	3rd of December 2018

Summary

In this report we describe the overall structure of the certified Behavior Tree execution engine and the associated specification extractor for the NuSMV/OCRA toolchain developed in the context of the CARVE project, as well as their formal relationship. The main goal of the report is to show that the execution engine and the specification extractor provide sufficient guarantees insofar as the correctness proof carried out on the extracted specification maps to the actual execution of Behavior Trees on the execution engine.

Contents

1 Introduction

The main goals of the CARVE project are:

- to propose *Behavior Trees* (BTs) as tools for modelling composable and reusable behaviors in the robotic domain;
- to develop the use of *formal methods* for increasing confidence in the correct execution of BTs.

In CARVE we wish to ensure that the "high level description" of the behavior of a robot given in terms of BTs is executed correctly. Towards this end we have designed and implemented the following tools:

- an *execution engine* for BTs, to be integrated into the SmartSoft framework, which implements the operational semantics defined in the deliverable D3.1. This tool is described in the D4.1 document;
- a *specification extractor*, i.e. an automatic tool to translate a BT into an equivalent Hierarchical Finite State Machine (HFSM) whose properties are amenable to verification by formal methods using established tools. This tool, described in D5.2 document, is based on a formal (operational) semantics for the execution of BTs — see document D3.2, "Syntax and Semantic of Behavior Trees for Robotics" wherein a translation from the BT formalism to the classical model of computation provided by HFSMs is given.

The main external tools involved in the process are the Coq proof assistant [3] (with associated program extraction capabilities), the NuSMV model checker [5] and the OCRA tool for contract refinement analysis [7].

During the development of the project we deemed necessary to introduce two extensions to the standard BT formalism:

- **reset messages**, ensuring correct interruption of software components;
- **contract annotations**, allowing us to express *constraints* that are assumed to be satisfied for correct task execution and *guarantees* that are ensured to be satisfied on successful task completion.

The rest of this report is structured as follows:

- Section 2 describes the formalization of the operational semantics for Behavior Trees on which the certified interpreter is based.
- Section 3 describes the translation process from Behavior Trees to Hierarchical Finite State Machines, implicitly expressed as SMV specifications.
- Section 4 describes the relationship between the two semantics.
- In the final section 5 we sketch some directions for future work.

The code described in this report can be found on the public Github repository `https://github.com/CARVE-ROBMOSYS/BTCompiler`

2 Operational semantics of Behavior Trees

In this section we describe in detail the implementation of the operational semantics of Behavior Trees in the Coq proof assistant which underlies the certified interpreter described in the companion document D4.1.

2.1 Description of the BT data type

The first step of the formalization process is the definition of a data type representing Behavior Trees in Gallina (the specification language used in Coq [4]). In a first phase of the project we proceeded in parallel along two different routes:

- a first approach using *binary trees*, i.e. trees having only nodes with binary branching;
- a second approach using trees with arbitrary (but greater than zero) branching.

The first implementation mainly served as a test-bed by presenting all the main difficulties of the formalization in a simplified setting (e.g. with no mutual inductive definitions needed). The second implementation, however, should be considered as the "final" one. It matches more closely the usual informal description of Behavior Trees [2], as well as the corresponding formalization by ALES-UTRC, as detailed in the D3.2 document. In what follows we shall only describe the second implementation.

To increase abstraction, we found useful to parameterize the modules implementing the BT data type over the following signature:

```
Module Type BT_SIG.
  Parameter skillSet: Set.
  Parameter skillName: skillSet -> string.
End BT_SIG.
```

This signature specifies a set whose members are the basic skills available to the BT designer and a function mapping each basic skill to its name (which is simply a string). Each skill is assumed to have a different name, although in the current implementation this constraint is not enforced at the type level.

Notice that these names are also used to build the NuSMV specification corresponding to a given Behavior Tree, hence they should also qualify as valid SMV identifiers. According to the NuSMV manual [6], this means that each of these strings may contain any sequence of characters starting with a character in the set `{A-Za-z}` and followed by a possibly empty sequence of characters belonging to the set `{A-Za-z0-9$#-}`. All characters and case in an identifier are significant.

Another important design decision we had to make regarding the BT data type is the following. A priori, there are two ways to represent trees with arbitrary branching in Coq:

- As a *mutual inductive type*: this is the classic definition in terms of trees and forests:

  ```
  Inductive tree: Set :=
  | node: symbol -> forest -> tree
  with forest: Set :=
  | fnil: forest
  | fcons: tree -> forest -> forest.
  ```

- As a *nested inductive type*: this definition uses the (polymorphic) `list` type from the Coq standard library:

  ```
  Inductive ntree: Set :=
  | leaf: symbol -> ntree
  | node: symbol -> list ntree -> ntree.
  ```

The upside of using nested induction is that in this case we can use "for free" the various functions and theorems about the `list` type which are defined in the Coq standard library. However, working with nested inductive types is also more involved: for instance Coq has no predefined induction scheme for these types, so that induction and recursion principles must be specified by hand. We direct the reader to Adam Chlipala's book [1, section 3.8] for an in-depth discussion of these points.

Another reason to use mutual induction with a specialized `btforest` type is that in this way we can enforce the constraint (explicitly specified in the D3.2 document) according to which a BT inner node always has at least one child, directly at the type level. This is clearly impossible to do with a nested inductive type, since regular Coq lists can be empty.

For these reasons we chose to formalize Behavior Trees using a pair of mutual inductive types, called `btree` and `btforest`. Their definition (parameterized on a module `X` of type `BT_SIG`) is as follows:

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

Notice how the `Child` constructor for the `btforest` type enforces BT forests to always have at least one member.

The various kinds of nodes and decorators are described by the `nodeKind` and `decKind` types. These are again inductively defined, the only nontrivial constructor being the one for Parallel nodes (which takes as a parameter its threshold):

```
Inductive nodeKind: Set :=
| Sequence: nodeKind
| Fallback: nodeKind
| Parallel: nat -> nodeKind.
Inductive decKind: Set := Not | IsRunning.
```

Algorithm 1: Pseudocode of a Sequence node with N children

```
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus = Running then
        return Running
    else if childStatus = Failure then
        return Failure
return Success
```

2.2 Basic operational semantics of BTs

In the file `basic.v` we formalize the basic operational semantics for Behavior Trees by defining a function (written in the Gallina programming language) whose execution reproduces the intended semantics of any given BT. In other words, we define what is usually called a *shallow embedding* of the BT language in the type theory underlying the Coq proof assistant.

We immediately remark that, since Gallina programs are guaranteed to always terminate, this automatically gives us a proof of termination for the BT execution engine that will be derived from the above definition by Coq's program extraction mechanism.

We chose to encapsulate the possible return values of a basic skill in the type `return_enum`, and to represent the return value of each skill at a certain instant of time with a term of type `skills_input`:

```
Inductive return_enum := Runn | Fail | Succ.
Definition skills_input := X.skillSet -> return_enum.
```

The heart of the formalization is the function `tick`, of type `btree -> skills_input -> return_enum`, defined by structural induction over the `btree` type. This function formalizes the execution semantics for Behavior Trees usually specified in the literature using informal pseudocode (as in the book [2], from which we took the algorithms listed below).

Let us break down the (somewhat involved) definition of the `tick` function according to the node type. The base cases of the recursion are straightforward:

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
  | Skill s => input_f s
  | TRUE => Succ
  [...]
  end
```

Let us focus on Sequence nodes. Their execution is usually described in terms of the pseudocode illustrated in Algorithm 1. This corresponds to the following fragment of the definition of `tick`:

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
[...]
  | Node k _ f => match k with
                  | Sequence => tick_sequence 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
```

Apart from translating the iteration into a recursion, the analogies between the two algorithms are clear.

Fallback nodes are similarly described in terms of Algorithm 2. This translates to the following Coq code fragment, very similar to the one above:

Algorithm 2: Pseudocode of a Fallback node with N children

```
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus = Running then
        return Running
    else if childStatus = Success then
        return Success
return Failure
```

Algorithm 3: Pseudocode of a Parallel node with N children and success threshold M

```
for i ← 1 to N do
    childStatus(i) ← Tick(child(i))
if Σ_{i:childStatus(i)=Success} 1 ≥ M then
    return Success
else if Σ_{i:childStatus(i)=Failure} 1 > N − M then
    return Failure
return Running
```

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
[...]
  | Node k _ f => match k with
                  | Fallback => tick_fallback f input_f
[...]
  end
with tick_fallback (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 => tick_fallback rest input_f
                        | Succ => Succ
                        end
       end
```

Finally, a Parallel node can be described in terms of the slightly more complex Algorithm 3, to be compared with the following Coq code fragment:

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
[...]
  | Node k _ f => match k with
                  | Parallel n =>
                    let results := tick_all f input_f in
                    if n <=? (countSucc results) then Succ
                    else if (len f - n) <? (countFail results) then Fail
                         else Runn
[...]
  end
with tick_all (f: btforest) (input_f: skills_input) :=
       match f with
       | Child t => cons (tick t input_f) nil
       | Add t1 rest => cons (tick t1 input_f) (tick_all rest input_f)
       end
```

where `countSucc` and `countFail` are auxiliary functions defined in the obvious manner.

Notice that in the above pseudocode the leaves of a parallel node are ticked in order according to their index. However, this feature is not part of the specification of the node; indeed, in the current implementation of the `tick` function the leaves of a parallel node are ticked *in reverse order* (i.e., starting from the last leaf and and going towards the first). As a rule, BT designers should not rely on any particular ticking order for Parallel nodes.

The final part of the `tick` function definition deals with decorators. Their execution reduces to a straightforward case analysis:

```
Fixpoint tick (t: btree) (input_f: skills_input) :=
  match t with
  [...]
  | Dec k _ t => match k with
                 | Not =>
                   match tick t input_f with
                   | Runn => Runn
                   | Fail => Succ
                   | Succ => Fail
                   end
                 | IsRunning =>
                   match tick t input_f with
                   | Runn => Succ
                   | Fail => Fail
                   | Succ => Fail
                   end
                end
  end
```

In the book [2] some other decorators are described, as for instance the *max-N-tries* or the *max-T-sec* decorators. These decorators are different from the preceding ones because they have an internal state. In our formalization, leaves of a BT cannot have a state; such algorithms must then be implemented as subtrees which manage the state via some basic skill implementing a suitable interface (for instance communication with a parameter server, seen as an external resource available to the BT).

2.3 Stream semantics

In the files `stream.v` and `preempt.v` we expand the basic approach to a formal operational semantics for BTs described in the previous subsection along two different directions.

In the first variation, that we call "stream semantics", the idea is to modify the `tick` function in order to make it operate on a *stream* of return values. This makes it possible to model inside Coq the process of *repeated ticking* of a Behavior Tree.

More in detail, the `skills_input` type is replaced in this setting by the type `input_stream` defined as follows:

```
Definition input_stream := Stream skills_input.
```

Here we use the implementation of streams in the standard Coq library as a coinductive type (lazy nonempty list). Then we can define three different ways to tick a BT:

- a (modified) `tick` function that evaluates a single tick by consuming values from the given input stream as needed (and discarding the resulting stream);
- a `tick2` function that evaluates a single tick by consuming values from the given input stream *and returning the resulting stream* (together with the BT output value);
- a `reptick` function that evaluates *a specified number of ticks* by consuming values from the given input stream and returns a list containing the results of the successive tickings.

It is interesting to note that a version of the `reptick` function with no upper bound on the number of ticks performed cannot be defined in Gallina (not even as a corecursive function on the input stream), since the corresponding corecursion would necessarily be unguarded.

2.4 Preemptible semantics

In the second variation, called the "preemptible semantics", we introduce the important notion of *reset messages*. In this case, each basic skill is assumed to come with a *reset function*, to be called every time the skill is *not* ticked in a Sequence or Fallback node. This is important, for instance, in robotics applications, in order to ensure that an action that has been started is not stopped half-way through its execution.

In our formalization, the reset functions for all the basic skills are assumed to be bundled together in the following function type:

```
Definition skills_reset := X.skillSet -> bool.
```

A term of this type maps each basic skill to the return value of the corresponding reset function. This value is interpreted as follows:

- a return value `true` means that the reset was successful and the execution can go on normally;
- a return value `false` means that the reset was *not* successful and the whole BT should return Failure.

Implementations are free to suppress this additional semantics by making their reset functions always return `true`.

The operational semantics itself is then modified as follows:

- the type of the `tick` function becomes

  ```
  Fixpoint tick (t: btree) (input_f: skills_input) (reset_f: skills_reset)
  ```

  including as a third parameter the tuple of reset functions;

- In Sequence and Fallback nodes, as soon as the return value of the node is certain, all the children which have not been ticked yet are sent a reset message, and the return value of the node is changed according to whether the reset was successful or not. For instance the Sequence node algorithm becomes

  ```
  tick_sequence (f: btforest) (input_f: skills_input) (reset_f: skills_reset) :=
    match f with
    | Child t => tick t input_f reset_f
    | Add t1 rest => match tick t1 input_f reset_f with
                     | Runn => let b := reset_forest rest reset_f in
                               if b then Runn else Fail
                     | Fail => let b := reset_forest rest reset_f in
                               if b then Fail else Fail
                     | Succ => tick_sequence rest input_f reset_f
                     end
    end
  ```

  where `reset_forest` is defined by mutual induction as follows:

  ```
  Fixpoint reset_bt (t: btree) (reset_f: skills_reset) :=
    match t with
    | Skill s => reset_f s
    | TRUE => true
    | Node _ _ f => reset_forest f reset_f
    | Dec _ _ t => reset_bt t reset_f
    end
  with reset_forest (f: btforest) (reset_f: skills_reset) :=
    match f with
    | Child t => reset_bt t reset_f
    | Add t1 rest => let x := reset_bt t1 reset_f in
                     andb x (reset_forest rest reset_f)
    end.
  ```

  Notice that the reset message is sent independently from the fact that the skill has already been ticked before or not: as in the basic semantics, the execution has no "memory" of the previous ticks.

- In Parallel nodes, as soon as the return value of the node is certain, all the children which have returned Running are sent (*in addition* to the previous tick) a reset message:

```
Fixpoint tick (t: btree) (input_f: skills_input) (reset_f: skills_reset) :=
  match t with
[...]
  | Node k _ f => match k with
                  | Parallel n =>
                    let results := tick_all f input_f reset_f in
                    if n <=? (countSucc results) then
                      let b := reset_running f results reset_f in
                      if b then Succ else Fail
                    else if (len f - n) <? (countFail results) then
                           let b := reset_running f results reset_f in
                           if b then Fail else Fail
                         else Runn
[...]
  end
```

where the helper procedure `reset_running` is defined by

```
Fixpoint reset_running (f: btforest) (l: list return_enum) (reset_f: skills_reset) :=
  match f with
  | Child t => match hd Fail l with
               | Runn => reset_bt t reset_f
               | _ => true
               end
  | Add t1 rest => let x :=
                       match hd Fail l with
                       | Runn => reset_bt t1 reset_f
                       | _ => true
                       end
                   in
                   andb x (reset_running rest (tl l) reset_f)
  end.
```

Notice that every call to `reset_running` always involves as `f` and `l` two lists having the same length, so that the calls to `hd` inside `reset_running` will, in fact, never fail.

3 From Behavior Trees to Hierarchical Finite State Machines

In this section we describe the details of the translation mechanism between Behavior Trees and Hierarchical Finite State Machines. The tool `mkspec` described in the D5.2 document is based on this mechanism.

3.1 The MicroSMV language

The formalization described in the previous section has the virtue of being close to the classical, pseudocode-style informal semantics for BTs already present in the literature. However, in order to bring into the game formal tools like model checkers and runtime monitor generators, we also need a way to translate a BT into a HFSM which tracks more closely the formal operational semantics for BTs defined in the D3.2 document.

There are various formalizations of Finite State Machines among Coq's user-contributed libraries[1]. However, it seemed difficult to adapt these implementations to our needs. Moreover, given that ALES-UTRC used the NuSMV model checker [6] to test their approach to design methodology for BTs, it was quite natural to choose the same tool as our model checking engine. The logical next step, then, was to leverage the SMV specification language to (implicitly) describe the HFSMs acting as target for the translation.

Unfortunately, there is currently no way to interface directly the Coq proof assistant with NuSMV. Thus, in order to be able to formally manipulate SMV specifications, we decided to embed into Coq (a restricted subset

[1] See for instance `coq-automata` `https://github.com/coq-contribs/automata`, `p-automata` `https://github.com/coq-contribs/pautomata`, `coq-fairisle` `https://github.com/coq-contribs/fairisle`.

of) the SMV language, in a similar vein to what Johnson-Freyd et al. did for TLA [8]. We call MicroSMV the resulting language. The SMV constructs to be included in it were specifically chosen in order to cover the hand-made SMV specifications for BTs provided by ALES-UTRC in the D3.1 document.

Notice that, with this approach, we were able to obtain the code which translates a BT to the corresponding FSM specification by program extraction. The code obtained in this manner enjoys all the usual guarantees provided by the use of total functional programming languages (assured termination, etc.).

The MicroSMV language has been briefly described in the D5.2 document. It is a highly simplified version of SMV, featuring only two basic types (booleans and symbolic enumerations), a reduced set of simple expressions (detailed below), `DEFINE` macros, `ASSIGN` constraints (used in SMV to specify initial states and transitions), and finally the support for parametric modules.

In the Coq file `micro_smv.v` we define a number of inductive types representing the (abstract) syntax of the language. In many cases these definitions were lifted directly from [6, Appendix C], with the appropriate simplifications. For instance, the type `sexp` corresponding to MicroSMV simple (or basic) expressions is defined as follows:

```
Inductive sexp :=
| BConst: bool_constant -> sexp
| SConst: symbolic_constant -> sexp
| Qual: qualid -> sexp
| Paren: sexp -> sexp
| Neg: sexp -> sexp
| And: sexp -> sexp -> sexp
| Or: sexp -> sexp -> sexp
| Equal: sexp -> sexp -> sexp
| Less: sexp -> sexp -> sexp
| Sum: sexp -> sexp -> sexp
| Count: sexplist -> sexp
| Case: scexp -> sexp
with sexplist :=
     | Sexp: sexp -> sexplist
     | AddSexp: sexp -> sexplist -> sexplist
with scexp :=
     | Cexp: sexp -> sexp -> scexp
     | AddCexp: sexp -> sexp -> scexp -> scexp.
```

Every operator is straightforward except for the `Case` constructor, which takes a (nonempty) list of pairs of simple expressions as argument. The idea is that the first element of the pair gives the boolean condition and the second element gives the value of the expression when the specified condition is true.

Notice that the resulting language has fairly weak typing rules: for instance a `count` expression expects a list of boolean values (and counts the number of true instances), but this domain restriction is not imposed in any way at the syntactic level.

An (abstract) *SMV module* is represented by a record type of the following form:

```
Record smv_module: Set :=
  { name: identifier;
    params: list identifier;
    vars: option varlist;
    ivars: option ivarlist;
    defs: option deflist;
    assigns: option asslist }.
```

In other words, an `smv_module` consists of a name, a list of formal parameters (which may be empty), and optionally:

- a list of (state) variables,
- a list of input variables,
- a list of `DEFINE` macros,
- a list of `ASSIGN` constraints.

A MicroSMV specification is simply a list of terms of type `smv_module`.

The file `spec_extr.v` contains the code needed to translate an abstract MicroSMV module into a textual representation (that is, a string) which can be directly feeded into the NuSMV model checker as input.

3.2 Automatic generation of the SMV specification

Now let us give a high-level overview of the translation process from Behavior Trees to SMV specifications, as implemented in the `bt2spec.v` file. As always in this document, we focus on the arbitrary-branching implementation, which is contained in the Coq module named `BT_gen_spec`.

- The *recursive* structure of Behavior Trees is reflected into the *hierarchical* structure of the corresponding SMV specification. In other words, the specification is organized in terms of SMV *modules with parameters*, similarly to a class hierarchy in object-oriented languages. See the SMV manual [6, subsections 2.3.10-11] for more information about SMV modules.
- Each kind of node (leaf nodes, Sequence nodes, etc.) corresponds to a particular SMV module declaration, and each instance of a given node inside a BT is mapped into an instance of the corresponding module in the SMV specification.
- The concrete mapping between BT nodes and SMV modules can be summarized as follows.
  - A basic skill (`Action` or `Condition` nodes in the usual BT jargon) is implemented using the fixed MicroSMV module `bp_skill_autonomous`[2], which (when translated to concrete SMV syntax) looks as follows:

    ```
    MODULE bt_skill()
    VAR
      output : { none, running, failed, succeeded };
      enable : boolean;
    IVAR
      input : { Runn, Fail, Succ };
    ASSIGN
      init(output) := none;
      next(output) := case
        !(enable) : none;
        input = Runn : running;
        input = Fail : failed;
        input = Succ : succeeded;
      esac;
    ```

  - A dummy leaf `TRUE` is implemented as the fixed MicroSMV module `bp_TRUE`, whose output is always `succeeded`.

    Notice that both modules `bp_skill_autonomous` and `bp_TRUE` have no formal parameters. This is a reflection of the fact that they are used to implement the "base cases" of the recursive data type `btree`.
  - A Sequence node with n children is translated to a MicroSMV module with n parameters and the following general structure:

    ```
    MODULE bt_sequence_n(btn, ..., bt1)
    VAR
      enable : boolean;
    DEFINE
      output := case
        btn.output = running | btn.output = failed : btn.output;
        bt(n-1).output = running | bt(n-1).output = failed : bt(n-1).output;
        ...
        bt2.output = running | bt2.output = failed : bt2.output;
        TRUE : bt1.output;
      esac;
    ASSIGN
      btn.enable := enable;
      bt(n-1).enable := btn.output = succeeded;
      ...
      bt1.enable := bt2.output = succeeded;
    ```

[2]The reason for the "`autonomous`" part will be explained below.

These modules are generated on demand by the `make_sequence` function[3].

- A Fallback node with n children is translated to a MicroSMV module whose general structure is similar to the one above, with the obvious changes in the switching logic. These modules are also generated on demand by the `make_fallback` function.
- A Parallel node with n children and threshold parameter k is translated to a MicroSMV module with the following general structure:

```
MODULE bt_parallelk_n(btn, ..., bt1)
VAR
  enable : boolean;
DEFINE
  true_output_count := count(btn.output = succeeded, ... , bt1.output = succeeded);
  fail_output_count := count(btn.output = failed, ... , bt1.output = failed);
  output := case
    k < true_output_count + 1 : succeeded;
    n < fail_output_count + k : failed;
    TRUE : running;
  esac;
ASSIGN
  btn.enable := enable;
  bt(n − 1).enable := enable;
  ...
  bt1.enable := enable;
```

  These modules are generated on demand by the `make_parallel` function. Notice that the threshold parameter does not appear as a parameter of the module; instead it is "baked in" directly inside the switching logic of the module.
- Finally, Decorator nodes are implemented using the fixed MicroSMV modules called `bp_not` and `bp_isRunning`. Both these modules have a single parameter, representing the single child of the node in the tree.

The entry point to the translation algorithm is provided by the `make_spec` function:

```
Definition make_spec (t: btree): list smv_module :=
  let needed := addmod t (empty_set modtype) in
  let modlist := make_mod_list needed true in
  app modlist (bp_tick_generator :: (make_main t "main") :: nil).
```

The specification is built in three steps:

1. The first step consists of building a list of all the module types needed for translating the given BT. The module types are elements of the inductive type `modtype`, defined by

   ```
   Inductive modtype :=
   | Skmod: modtype
   | TRUEmod: modtype
   | Seqmod: nat -> modtype
   | Fbmod: nat -> modtype
   | Parmod: nat -> nat -> modtype
   | Notmod: modtype
   | Runmod: modtype.
   ```

   The first parameter of the `Seqmod`, `Fbmod` and `Parmod` constructors is simply the number of childen of the node. The second parameter of `Parmod` is the threshold of the corresponding Parallel node.

2. In the second step, the `make_mod` function is recursively applied to each element of the module list, in order to generate the corresponding `smv_module` term. `make_mod` is defined by a simple pattern matching:

[3]The reader may wonder why the formal parameters were named in "reverse order", that is starting from `btn` and decreasing towards `bt1`. This was done in order to streamline the recursive generation of the module: indeed in this way the base case for a recursion over a `btforest` (which is the rightmost children, always named `bt1` in this scheme) coincides with the base case for a recursion over the type of (positive) natural numbers.

```
Definition make_mod (t: modtype) (aut: bool): smv_module :=
  match t with
  | Skmod => if aut then bp_skill_autonomous else bp_skill
  | TRUEmod => bp_TRUE
  | Seqmod l => make_sequence l
  | Fbmod l => make_fallback l
  | Parmod n l => make_parallel n l
  | Notmod => bp_not
  | Runmod => bp_isRunning
  end.
```

(see below for the meaning of the `aut` parameter).

3. In the final step, the list of MicroSMV modules obtained at the previous step is completed with the addition of the `bp_tick_generator` and `main` modules.

   The purpose of `bp_tick_generator` module is simply to send a tick event to the HFSM implementing the given BT at each time step of the simulation.

   ```
   MODULE bt_tick_generator(top_level_bt)
   ASSIGN
     init(top_level_bt.enable) := TRUE;
     next(top_level_bt.enable) := !(top_level_bt.output = none);
   ```

   Notice that the execution of the HFSM implementing a BT is completely synchronous: a tick propagates from the root to the leaves and generates a return value for the whole tree in a single time step.

   Finally, the only job left for the `main` module is to instantiate the preceding modules according to the hierarchy derived from the recursive structure of the given BT. The names given to the various instances of each module are simply the strings contained in the various `Node` and `Dec` constructors of the `btree` data type (see subsection 2.1).

The algorithm described above produces a standalone SMV specification, which can then be feeded directly into NuSMV for formal verification purposes. However, by default `mkspec` will not produce such a specification. Instead, it will produce an SMV module ready to be interfaced with the OCRA verification tool, as a possible implementation for the `BT_FSM` component described in the companion OSS (OCRA System Specification) file. We direct the reader to the D3.1 and D5.2 document for a description of the role played by OCRA in the CARVE project.

4 Relationship between the two semantics

In this section we explain how to bridge the gap between the operational semantics of BTs underlying the certified interpreter (described in section 2) and the operational semantics for the corresponding HFSM which can be subjected to model checking using NuSMV (described in section 3).

Ideally, the gap could be bridged by developing a mechanical proof inside Coq. Indeed, one can surely define inside Coq an operational semantics (i.e., a notion of execution) for HFSMs specified in the MicroSMV language, and then try to formally prove that for any term `t` of type `btree`, the result of the execution of the corresponding HFSM coincides with the result of the function `tick` applied to `t` (for the same output values of the basic skills).

However, such a proof would be rather pointless, since in practice the model checking step we perform is *not* done by (the code extracted from) the operational semantics for MicroSMV defined above. Instead, the model checking is performed by the totally unrelated NuSMV code. This code is not formally certified anyway[4], and so nothing guarantees that it correctly implements the particular operational semantics we defined.

The only way to reconcile the two semantics in a fully formal way would be, actually, to develop a (formally verified) model checker in Coq and use that model checker to perform validation of (the HFSM equivalent to) a BT. Obviously this route was out of the question for the CARVE project, at the very least for reasons of time and manpower needed.

So we shall content ourselves with an informal proof of coherence between the (formal) semantics underlying the BT interpreter and the (informal) semantics employed by NuSMV for model checking of the corresponding HFSMs.

[4] Although it is surely very well tested "on the field" by decades of use in academia and industry.

Let us show, then, that the HFSMs obtained by using the `mkspec` tool really have the same behavior of the BT execution engine on each given BT. We shall proceed by structural induction on the type of Behavior Trees, as it was illustrated in subsection 2.1.

Let us start with the base cases of the induction (i.e. the possible leaf nodes of a BT).

- For a TRUE leaf node, the result follows trivially from the fact that both the `tick` function and the MicroSMV module `bp_TRUE` always return success.
- Now let s be a basic skill. Then the result of the `tick` function is just the returning value of skill s. In terms of HFSMs, a BT consisting of a single leaf is mapped to to the following SMV specification:

```
MODULE bt_skill()
VAR
  output : { none, running, failed, succeeded };
  enable : boolean;
IVAR
  input : { Runn, Fail, Succ };
ASSIGN
  init(output) := none;
  next(output) := case
  !(enable) : none;
  input = Runn : running;
  input = Fail : failed;
  input = Succ : succeeded;
  esac;

MODULE bt_tick_generator(top_level_bt)
ASSIGN
  init(top_level_bt.enable) := TRUE;
  next(top_level_bt.enable) := !(top_level_bt.output = none);

MODULE main()
VAR
  s : bt_skill;
  tick_generator : bt_tick_generator(s);
```

whose flattened version reads

```
MODULE main()
VAR
  s.output : { none, running, failed, succeeded };
  s.enable : boolean;
IVAR
  s.input : { Runn, Fail, Succ };
ASSIGN
  init(s.output) := none;
  next(s.output) := case
    !(s.enable) : none;
    s.input = Runn : running;
    s.input = Fail : failed;
    s.input = Succ : succeeded;
  esac;
  init(s.enable) := TRUE;
  next(s.enable) := !(s.output = none);
```

The execution of this FSM is straightforward: at odd time steps, the variable `enable` is true and `output` is `none`, whereas at even time steps `enable` becomes false and `output` coincides with the skill's return value. Overall, the result is obviously consistent with the result value of the `tick` function.

Now let us go on with the inductive part of the proof. We analyze separately each possible inner node of a BT and prove that, under the assumption that its children verify the theorem, the same holds for the return value of that node.

- *Sequence nodes.* For the sake of clarity let us consider a *binary* sequence node (the argument for longer sequences is analogous). A BT containing a Sequence node with two children, call them `c1` and `c2`, will give rise to a specification containing the module

```
MODULE bt_sequence_2(bt2, bt1)
VAR
enable : boolean;
DEFINE
output := case
bt2.output = running | bt2.output = failed : bt2.output;
TRUE : bt1.output;
esac;
ASSIGN
bt2.enable := enable;
bt1.enable := bt2.output = succeeded;
```

  which is then instantiated in the `main` module with a variable declaration of the form

```
seq: bt_sequence_2(c1,c2);
```

  After the flattening step, the resulting FSM is going to contain, among the others, the following declarations:

```
VAR
  c1.output : {none, running, failed, succeeded};
  c1.enable : boolean;
  c2.output : {none, running, failed, succeeded};
  c2.enable : boolean;
  seq.enable : boolean;

DEFINE
  seq.output := case
    (c1.output = running | c1.output = failed) : c1.output;
    TRUE : c2.output;
  esac;

ASSIGN
  c2.enable := c1.output = succeeded;
  c1.enable := seq.enable;
  init(seq.enable) := TRUE;
  next(seq.enable) := !(seq.output = none);
```

  This means that, at a time step when the `seq` node is enabled, the value of its output will be computed (using the same algorithm used by the `tick` function) on the basis of the values of the two variables `c1.output` and `c2.output`. This ensures that the final value of the variable `seq.output` coincides with the result of the `tick` function operating on the same node.

- *Fallback nodes.* Again, let us focus on the binary case. A generic binary fallback module looks as follows:

```
MODULE bt_fallback_2(bt2, bt1)
VAR
  enable : boolean;
DEFINE
  output := case
    bt2.output = running | bt2.output = succeeded : bt2.output;
    TRUE : bt1.output;
  esac;
ASSIGN
  bt2.enable := enable;
  bt1.enable := bt2.output = failed;
```

which is then instantiated with a declaration of the form

```
fb: bt_fallback_2(c1,c2);
```

After the flattening step, the following declarations are obtained:

```
VAR
  c1.output : {none, running, failed, succeeded};
  c1.enable : boolean;
  c2.output : {none, running, failed, succeeded};
  c2.enable : boolean;
  fb.enable : boolean;

DEFINE
  fb.output := case
    (c1.output = running | c1.output = succeeded) : c1.output;
    TRUE : c2.output;
  esac;

ASSIGN
  c2.enable := c1.output = failed;
  c1.enable := fb.enable;
  init(fb.enable) := TRUE;
  next(fb.enable) := !(fb.output = none);
```

and reasoning similarly to the Sequence case, the same conclusion follows.

- *Parallel nodes.* Consider now a parallel node with n children and threshold k. The corresponding SMV module has been shown in subsection 3.2. Once flattened, it gives rise to declarations of the form

```
VAR
  c1.output : {none, running, failed, succeeded};
  c1.enable : boolean;
[...]
  cn.output : {none, running, failed, succeeded};
  cn.enable : boolean;
  par.enable : boolean;

DEFINE
  par.output := case
    k < par.true_output_count + 1 : succeeded;
    n < par.fail_output_count + k : failed;
    TRUE : running;
  esac;
  par.fail_output_count := case
    cn.output = failed : 1;
    TRUE : 0;
  esac +
  [...] +
  case
    c1.output = failed : 1;
    TRUE : 0;
  esac;
  par.true_output_count := case
    cn.output = succeeded : 1;
    TRUE : 0;
  esac +
  [...] +
  case
    c1.output = succeeded : 1;
```

```
    TRUE : 0;
  esac;

ASSIGN
  c1.enable := par.enable;
[...]
  cn.enable := par.enable;
  init(par.enable) := TRUE;
  next(par.enable) := !(par.output = none);
```

Here all the children are enabled at the same time step, corresponding to the call to `tick_all` in the definition of the `tick` function for Parallel nodes. The various results are then combined using the same algorithm in both cases, and this guarantees once again that exactly the same result is arrived at.

- *Decorator nodes.* This case also follows easily by comparing the execution of the fixed modules `bp_not` and `bp_isRunning` with the definition of the `tick` function on decorator nodes that we showed in subsection 2.2.

Thus we can conclude that the execution of the generated HFSMs faithfully reproduces the execution algorithm for BTs implemented by the `tick` function.

5 Future developments

Our formalization of Behavior Trees can be improved in many ways; let us briefly discuss some of the most important ones.

Using sharper (dependent) types. The current code only uses classical (i.e., non-dependent) polymorphic typing. In many instances it would have been useful to incorporate at a deeper level the "correct by definition" philosophy enabled by the use of dependently-typed programs defined inside a proof assistant [1]. Some examples of these situations are listed below.

- A Parallel node whose parameter (success threshold) has a value greater than the number of its children is meaningless and should not appear in any well-formed Behavior Tree. Using a dependently-typed constructor, it is possible to enforce a constraint of the form "threshold $\leq$ number of children of the node" directly at the type level, in order to obtain a sharper characterization of the BT data type. Without this sharpening, for instance, it is impossible to prove that a Parallel node with a single child (and any positive value for the threshold) is semantically equivalent to the child tree itself.
- Another possible sharpening has to do with the length function for BT forests. Its return type is `nat`, the type of natural numbers, whereas a sharper specification would use the type of *positive* natural numbers, since every BT forest has at least one member. This would simplify the definition of various functions in the file `bt2spec.v`, by eliminating irrelevant cases (patterns that will never match).
- As briefly noted at the end of section 2, the implementation of `reset_running` would also benefit from a sharper typing for the formal parameters `f` and `l`.

After some internal discussion we decided not to go the dependently-typed route for now, leaving the implementation of sharper specifications to a future extension of this work.

Incorporating nodes with memory. A popular extension to the classic Behavior Tree syntax is provided by *nodes with memory*, also known as *starred nodes* (see e.g. [2, subsection 1.3.2]). For instance, a Sequence node with memory remembers whether each child has returned Success at the previous tick, and avoids the re-execution of that child until the whole node reaches a Success or Failure status. This kind of behavior brings into the game a whole new set of problems, having to do with the necessity of introducing a notion of *state* inside the BT execution engine.

In the D3.2b document (addendum to D3.2) the reader can find a proposal for an SMV formalization of nodes with memory. Integration of such nodes into the BT interpreter requires a significant effort which could be the target of future work.

Parametric skills. Another useful extension to the basic BT formalism is the possibility of using *parametric skills*, that is basic skills accepting one (or more) parameter(s). This extension will be also left to future developments of this project.

References

[1] Chlipala, A., *Certified Programming with Dependent Types*. https://adam.chlipala.net/cpdt.

[2] Colledanchise, M., and Ögren, P., *Behavior Trees in Robotics and AI*. https://arxiv.org/abs/1709.00084

[3] The Coq proof assistant. https://coq.inria.fr/

[4] The Coq Reference Manual. https://coq.inria.fr/distrib/current/refman/

[5] The NuSMV 2.6 model checker. https://nusmv.fbk.eu

[6] Cavada R., Cimatti A., Jochim C. A., Keighren G., Olivetti E., Pistore M., Roveri M. and Tchaltsev A., NuSMV 2.6 User Manual, 2010. https://nusmv.fbk.eu

[7] Cimatti A., Dorigatti M. and Tonetta S., OCRA: Othello Contracts Refinement Analysis, Version 1.3. https://es.fbk.eu/tools/ocra/

[8] Johnson-Freyd, P., Hulette G.C., and Ariola Z. M., *Verification by Way of Refinement: A Case Study in the Use of Coq and TLA in the Design of a Safety Critical System*. In Critical Systems: Formal Methods and Automated Verification. Springer, Cham, 2016. 205-213.