A Generation of the Transition System to Model-Check

This appendix is not meant to be read as part of the TACAS review process. It is not required to follow the results presented in the main part of the paper, and is considered supplementary material. It is made accessible via DOI link over Zenodo:

https://doi.org/10.5281/zenodo.10004424

This appendix describes details about the generation of the transition system $S_{E \sqcup P}$, from a C++ *behavior planner (BP)* and an SMV *environment model (EM)*, with the intention to facilitate reproducibility of the results. The artifact published with the paper (cf. the material on Zenodo) can be used out of the box to reconstruct the exact outcome presented in the paper, as well as to inspect the effects of some minor changes to the BP. However, actually deploying the proposed workflow for further investigation, and, e. g., model-checking a completely different piece of C++ code, or conducting in-depth changes to the EM requires more background knowledge, which we present below. We trust that these explanations, together with the full executable toolchain, and the example BP code to play around with, can provide an adequate understanding of the process.

A.1 Datatypes and General Interface Between EM and BP

We assume that the BP is an individual software component implemented in C++ with an interface to the surrounding software stack, i. e., perception inputs on one side and output towards actuation on the other. The EM is an additional component given in SMV, which is part of the MC toolchain only. It simulates the behavior of the surrounding environment, i. e., road topology, non-ego traffic participants etc., and incorporates ego behavior in a closed-loop manner, by providing input signals for the BP, and reacting on its output signals. Thus, it replaces and mocks the surrounding software stack including sensing of and reacting to the environment (cf. Fig. 2 in the paper).

We call BP variables used in a C++ sense *signals* to distinguish them from variables within a transition system. Signals, as opposed to variables, can have composite or array types. (nuXmv does support arrays, but only of primitive types, therefore, we do not use this feature.) Let N be a (semi-formal) set of all allowed member, type or variable names in C++ (i. e., alpha-numeric strings which do not start with a number; possibly including `::` to fully qualify `enum class` definitions or denote namespaces). The set $\mathbb{T}$ of (supported) types for signals in a C++ program is defined by:

- $\{void, int, float, bool\} \subseteq \mathbb{T}$ (primitive built-in types);
- $\forall i \in \mathbb{N}, e_1, \ldots, e_i \in N \colon enum[e_1, \ldots, e_i] \in \mathbb{T}$ (primitive enum types);
- $\forall i \in \mathbb{N}, t_1, \ldots, t_i \in \mathbb{T}, n_1, \ldots, n_i \in N \colon ((t_1, n_1), \ldots, (t_i, n_i)) \in \mathbb{T}$ (composite types given by `struct` or `class` definition);
- $\forall i \in \mathbb{N}, t \in \mathbb{T} \colon t^i \in \mathbb{T}$ (arrays of fixed size i).

Each type $t \in \mathbb{T}$ is identified by a name from N in C++. When translating the code towards a transition system for nuXmv, all signals need to be mapped to variables of one of the available primitive types. We map the primitive C++ types directly to their respective counterparts in nuXmv: *bool* $\Rightarrow$ `boolean`, *enum*[...] $\Rightarrow$ `enum[...]`, *int* $\Rightarrow$ `int`, *float* $\Rightarrow$ `real`[7]. Composite types are unfolded such that all contained primitive sub-signals are given an explicit variable in nuXmv. For example, `struct X { int i; float f; }; ...; X x; ...` defines a type $\texttt{X}\widehat{=}((int, i), (float, f)) \in \mathbb{T}$ and uses it for some signal `x`. In the transition system, this would yield two variables `x.i` and `x.f` of types `int` and `real`, respectively. Arrays are unrolled into explicit signals for each element; these signals are in turn mapped to primitive variables as above. For example, `std::array<X, 3> xarr;` would be unrolled into three signals `X xarr___609___; X xarr___619___; X xarr___629___;`.

This process yields the set of variables X_P of the BP transition system S_P. They can either be *internal* (i. e., used only for storage of internal state of the BP) or *input variables* $X_P^I \subseteq X_P$ or *output variables* $X_P^O \subseteq X_P$ which are "fed" or "read" by the EM (note that a single composite signal of the BP may induce internal, input and output variables at once at transition system level). EM variables (X_E) can, in principle, all be used as input or output for the BP. They, therefore, provide a *generic interface* which a BP can link to as desired. This allows to use a single EM for each occurring BP during development, as long as no fundamentally new interface requirements arise (i. e., as long as the EM is "suitable" for the BP, see below).

The specific interface between S_E and S_P is defined as follows. For variable $x \in X_P^I$ ($x \in X_P^O$) we denote by $\mathcal{I}_{P,E}(x) \subseteq X_E$ ($\mathcal{O}_{P,E}(x) \subseteq X_E$) the set of EM variables that can provide the input for x (expect the output given by x from the BP). $\mathcal{I}_{P,E}(x)$ and $\mathcal{O}_{P,E}(x)$ should usually either be empty or contain a single variable, otherwise different variables would be storing the same data. S_E is called *suitable* for S_P iff $\forall x \in X_P^I : \mathcal{I}_{P,E}(x) \neq \emptyset$ and $\forall x \in X_P^O : \mathcal{O}_{P,E}(x) \neq \emptyset$. Given S_E, S_P suitable for each other, the single element contained in $\mathcal{I}_{P,E}(x)$ ($\mathcal{O}_{P,E}(x)$) on EM side is then to be connected to x on BP side for all input and output variables x. Populating $\mathcal{I}_{P,E}(x)$ and $\mathcal{O}_{P,E}(x)$ requires semantic knowledge and needs to be done by a human expert; doing this with preferably little effort on developers' side is a central problem to solve. We do this by tagging signals in the BP with the name of their respective counterpart in the EM. These tags are given as comments in the C++ code, written behind the definition or usage of a signal, cf. Tab. 2 and Sec. A.4.

A.2 Interface from the Presented BPs Towards the Presented EM

The interface from a given BP towards the EM is generally defined by how it is "cut out" of the surrounding software stack. For the actual $\langle 1 \rangle$ and the mock BP $\langle 2 \rangle$ discussed in this paper, this is the interface required to be served:

[7] Note that this is already an abstraction, because, e. g., `int` in C++ is not a mathematical integer, and `float` is not a rational number.

Table 2: Most important tags used to control the translation of the BP. There are more tags which can be played around with in the artifact (cf. `vfm-options-planner.txt` and `planner.cpp` in the artifact), but their usage in the presented setup may not be fully supported.

	Tag	Effect

(1)	`// #vfm-begin`
	`<code>`
	`// #vfm-end`
		`<code>` is included into MC.

(2)	`// #vfm-cutout-begin`
	`<code>`
	`// #vfm-cutout-end`
		`<code>` is excluded from MC, even if within (1).

(3)	`// #vfm-gencode-begin`
	`<code>`
	`// #vfm-gencode-end`
		`<code>` is entrance point for execution w. r. t. MC.

(4)	`...x // #vfm-aka[[ literal(y) ]]`	Set y $\in \mathcal{I}_{P,E}$(x) if x $\in X_P^I$ or y $\in \mathcal{O}_{P,E}$(x) if x $\in X_P^O$. If x's type is composite, unroll to primitive types and perform the process to all arising sub-signals. Effectively, the EM is told to use `y` to either feed the BP's input `x` or to collect the BP's output `x`, respectively. (For composite `x, y`, all primitive sub-variables need to exist on EM side; both inputs and outputs can be contained within one signal, then; cf. `agent.v` and `agent.a` in the mock BP.)
(5)	`...x // #vfm-tal[[ a..b ]]`	Force the target type of an integer BP variable `x` to be the nuXmv integer range type `a..b` instead of `int`. (Support in the artifact is limited for this feature; in the mock planner it is used only to guide the intermediate representation, while nonetheless mapping to `int` in the end.) Note that the AKA and the TAL tags can be used in a fairly liberal syntax which is not completely described here. For example, in addition to putting them directly behind the variable name in question, it is allowed for readability to have a semicolon or comma in between, possibly including an initialization, e. g., `int x = 3; // #vfm-tal...`. Also, `a`, `b` and `y` can be formulas whose syntax is fairly straightforward, e. g., `m+4*n`, but also not further detailed here.
(6)	`// #vfm-inject[[ <formula> ]]`	Inject (non-standard) code to be used at C++ parsing time, which is not part of MC and, therefore, is not itself derived from C++. For example, `// #vfm-inject[[ @f(++) {x} { x = get(x) + 1 } ]]` injects a function `++x` for incrementing variable `x`; `// #vfm-inject[[ @f(ops::nop) {x} { 0 } ]]` injects a "nop" function doing nothing. Now, these functions are recognized by the parser when used in the C++ code and replaced by the respective logic. `// #vfm-inject[[ @x = y ]]` can be used to set the value of some variable `x` to `y`. Note that the injects are performed **before** parsing, not at the position they occur. Use instead (9) to add a string to some specific position in the code.
(7)	`// #vfm-option[[ general_mode << <option> ]]`	Allows to control the verbosity of the output and the creation of additional files. Set `<option>` to either `regular` or `debug`.
(8)	`// #vfm-option[[ optimization_mode << <option> ]]`	Optimizes the parsed BP code using a set of simplification rules, before converting it into a transition system. `<option>` can be `all` or `inner_only` (default; optimizes only code generated during parsing, **not** the C++ code). (The third option `none` will not work in the published setup.)
(9)	`// #vfm:<code>`	Adds `<code>` at the specified position to the C++ code. Can be used to sneak in code for MC that is ignored by the actual C++ compiler.

- (Input) Current **ego speed** in $^{m}/_{s}$.
- (Input) "Gap" data, i. e., information about lead vehicles to the front and rear on ego's own lane and its (up to) two neighboring lanes, cf. Fig. 3:
  (0) `gaps[ActionDir::LEFT]`: the fast lane next to ego (if present),
  (1) `gaps[ActionDir::CENTER]`: ego's own lane, and
  (2) `gaps[ActionDir::RIGHT]`: the slow lane next to ego (if present).

  Each gap provides the following information for the front and rear vehicle:
  - `i_agent_[front/rear]`: Id of the closest car to the [front/rear] of ego.
  - `s_dist_[front/rear]`: Distance from ego to this car.
  - `v_[front/rear]`: Velocity of this car.
  - `a_[front/rear]`: Acceleration of this car.
  - `turn_signals_[front/rear]`: Direction of the turn signals of this car (`ActionDir::left`, `ActionDir::right` or `ActionDir::none`).
- (Output) Long. control: desired acceleration $a \in \{-8, \dots, 2\}$ $^{m}/_{s^2}$ – *only* $\langle 2 \rangle$
- (Output) Lat. control:
  - Change towards fast lane (Boolean).
  - Change towards slow lane (Boolean) – *only* $\langle 2 \rangle$.
  - *(Or else follow the current lane.)*

A.3 Important Encodings in the EM

Lane Encoding. The EM uses three Boolean variables `lane_b1`, ..., `lane_b3` to store the lane association (either on a lane or between two lanes) of a car, e. g., `veh[1].lane_b2` and `veh[1].lane_b3` are set to true in Fig. 1 in the paper, but not `veh[1].lane_b1`. We use `DEFINE`s to create helper variables `veh[i].lane_1`, `veh[i].lane_12`, `veh[i].lane_2` etc. which track for each possible lateral position if a vehicle or ego is in that position or not. Note that this encoding replaced an earlier version based on an `int` variable holding the number of a car's lane, since it proved to be significantly more efficient this way.

Gap encoding. The published EM contains two versions of tracking which cars are in the three gaps around ego (cf. Sec. 3.2). One (the first one, chronologically) uses `ASSIGN`s as basic structure for implementation in nuXmv. Here, we first find the car closest to ego (in front or rear), and then fill in all the information of this car into the respective position in the gap. This version was later replaced by an `INVAR`-based version where the *value* of the closest distance to ego is calculated first, and the respective car is found subsequently based on which car has this distance to ego. Being less intuitive, the second version proves more efficient at least for two and more non-ego cars. It is currently the only one used, but the other can be triggered by setting `THRESHOLD_FOR_USING_ASSIGNS_IN_GAP_STRUCTURE` to a value larger than 0.

A.4 Enriching the C++ Code with Tags

The BP is developed in C++ and needs to be parsed from plain source code, largely in a "read-only" manner to minimize negative impact on development speed. In the ADAS project context, we found it acceptable to enrich the code with tags given as comments (cf. Tab. 2), as long as the "every-day" effort for their maintenance is negligible (cf. Sec. 4.3 for a discussion on this topic). The tags need to be provided with a background knowledge about the EM, but they pose mostly a one-time effort, as argued in Sec. 3.1. Most importantly, the tags specify $\langle 1 \rangle$ which parts of the code are supposed to be model-checked at all, using the `// #vfm-begin`... `// #vfm-end` tags; $\langle 2 \rangle$ where the entry point is, using the `// #vfm-gencode-begin` ... `// #vfm-gencode-end` tags; and $\langle 3 \rangle$ how the C++ signals connect to EM variables, using the `// #vfm-aka[[ literal(...) ]]` tag. For example,

```
int velocity // #vfm-aka[[ literal(ego.v) ]]
```

would connect the BP signal `velocity` to the EM variable `ego.v`. Formally, this would result in $\texttt{ego.v} \in \mathcal{I}_{P,E}(\texttt{velocity})$ if $\texttt{velocity} \in X_P^I$ or $\texttt{ego.v} \in \mathcal{O}_{P,E}(\texttt{velocity})$ if $\texttt{velocity} \in X_P^O$. Connecting a full composite signal, say of type $\texttt{Agent} \widehat{=} ((int, \texttt{v}), (int, \texttt{a})) \in \mathbb{T}$, with `v` being input and `a` output for the BP, to respective `ego...` variables, can be accomplished by:

```
Agent agent // #vfm-aka[[ literal(ego) ]]
```

This would trigger unrolling the signal to primitive sub-signals and connect each of them to variables of the respective names in the EM, assuming they exist, i. e., $\texttt{ego.v} \in \mathcal{I}_{P,E}(\texttt{agent.v})$ and $\texttt{ego.a} \in \mathcal{O}_{P,E}(\texttt{agent.a})$.

A.5 Parsing the C++ Code

For C++ processing we do not use a full C++ compiler or abstract syntax tree generator like *clang*, but a lightweight custom parser which can handle a supported subset of the C++ syntax.[8] It can handle all the usual program structures

[8] The main reasoning for this decision was the observation that MC can be applied to specific sub-parts of a software stack only, and using a custom tool allows to $\langle 1 \rangle$ more freely specify these sub-parts, $\langle 2 \rangle$ react more specifically on errors such as unsupported code blocks (which are probably unavoidable when model-checking a language as rich as C++), and $\langle 3 \rangle$ provide a more profound analysis of how code is involved in violations to the developers. There are also good reasons to use a full official C++ parser, though, and discussions are ongoing whether or not to switch in future. Particularly, encountering unforeseen unsupported syntax during development is an incalculable risk which could require time-consuming amendments to the parser. (Note that this could still happen, in different flavors, when using a full parser.) Probably the most considerable limitation is that it is currently not possible to follow calls to member functions (including constructors/destructors) relative to an object, as well as using custom operators on composite types. Also, "hard-to-

within functions, including calls to other free functions (in the published artifact leaving out loops and recursion, for minor technical reasons), `struct` and `class` definitions including inheritance, `enum` and `enum class` definitions, variable and constant declarations and definitions, pointers etc. We do not check for syntactical correctness, but assume this is done separately by a regular compiler.

The files containing the C++ code to check are listed in an additional text file (`vfm-includes-planner.txt`, in the artifact) and need to be specified in the order according to their usage (e. g., a `struct` definition needs to appear before it being used). In future, this could be avoided by following the C++ "`include`"s. The thus specified parts of the BP code are pasted together and build up the snippet to model-check. In the artifact, we additionally include a file called `vfm-options-planner.txt` containing options for the parser, cf. Tab. 2 in the appendix for an overview of what the options do.

A.6 Generation of S_P and $S_{E \sqcup P}$

For the generation of the intermediate representation, (4) in Fig. 1, the C++ BP code is first parsed and enriched by the interface information from (2) and (3). It is then further processed by inlining function calls, unfolding composite and array types, and finally establishing the interface between EM and BP. The details of this process can be observed in the published artifact.

A noteworthy peculiarity is the *unfolding of arrays* since it is fairly elaborate, while a simpler solution does not seem to be at hand for now. Since we use primitive variables to emulate arrays, it is not straight-forward to access an array element by using a non-constant index. As mentioned above, the access to an array `arr` at the constant position 0, `arr[0]`, is emulated by introducing a variable `arr___609___` which holds the value from `arr` at position 0. However, this approach is obviously not directly generalizable to a variable array access, like `arr[i]`, since the element to access depends on the value of `i`. Having to deal, in practice, with fairly small arrays, we decided to introduce a variable `arr___6i9___` and some additional logic at the access position to emulate the correct behavior. Assume a read access on an array with, say, three elements:

```
if (arr[i] > 4) {
   // do something
}
```

We would then introduce the additional logic as follows:

parse" C++ stuff like macros, templates, "`auto`" type deduction, and probably many more, are currently not supported. Pointers are supported in a "reference" way, i. e., there is no support for `nullptr` or pointer arithmetic. On the other hand, the limited syntax has proven sufficient for so far 5 use cases tried out in the scope of the ADAS project (including the one presented here), with so far none of interest being unparsable. In some cases minor reformulations of the BP code were necessary.

```
if (i == 0) {
   arr___6i9___ = arr___609___;
} else if (i == 1) {
   arr___6i9___ = arr___619___;
} else if (i == 2) {
   arr___6i9___ = arr___629___;
}

if (arr___6i9___ > 4) {
   // do something
}
```

In the context of a write access, such as this:

```
arr[i] = 4;
```

We would introduce this logic, replacing the original line:

```
if (i == 0) {
   arr___609___ = 4;
} else if (i == 1) {
   arr___619___ = 4;
} else if (i == 2) {
   arr___629___ = 4;
}
```

A nested array access with variables is resolved by first flattening the nested access via usage of temporary variables, for example, this code:

```
var = x[y[z[i.var[m]].var[j[k]]].var].var;
```

Would (essentially) be translated into this code, where the `ti` are new temporary variables, to which then the above process for flat array access can be applied:

```
t2 = i.var[m];
t3 = j[k];
t1 = z[t2].var[t3];
t0 = y[t1].var;
var = x[t0].var;
```

This approach is only suitable for "small" arrays; for "larger" ones, a better solution might be to keep a dedicated variable for the specific access updated at all time (though write access still needs to be handled separately). We did not investigate where the threshold lies. The arrays we deal with in the real code of the BP so far comprise no more than five elements, which we found to be sufficiently covered by the suggested solution.

The completed intermediate representation of the BP is translated into K2 language (`planner.k2` in Fig. 2; cf. also Alg. 3 for an example translation from actual C++ code), and from there into SMV (`planner.k2.smv`). It is then composed with the EM, by creating an additional integration file in SMV (5), which is depicted in Alg. 2. It constitutes the full transition system $S_{E \sqcup P}$, and, thus, the final model-checkable code which can be processed by nuXmv. Specifications can be given in this file, too. The full workflow *actually* contains expressing specifications at the C++ BP level using tags, and automatically inserting them from there into the integration file (which has some benefits, such as more customized error messages if the specifications do not match the code). However, this functionality is not available in the artifact.

Algorithm 1: Major part of the mock behavior planner. Preamble and main function omitted for space reasons. `static_cast`s removed to increase readability. Lines tagged “K2” indicate the logic which is shown in the K2 example in Alg. 3. Full compilable version available as supplementary material.

```
[...] // Includes etc.
// #vfm-begin
enum class ActionDir { LEFT = 0, CENTER = 1, RIGHT = 2, NONE = 3 };

struct Gap {
   int i_agent_front{-1}; // #vfm-tal[[-1..2]]  // The type abstraction layer (TAL) collects information...
   int s_dist_front{256}; // #vfm-tal[[0..256]] // ...about the model checker types, initial values etc.
   int a_front{0};        // #vfm-tal[[-8..6]]
   int i_agent_rear{-1};  // #vfm-tal[[-1..2]]
   int s_dist_rear{256};  // #vfm-tal[[0..256]]
   int v_rear{0};         // #vfm-tal[[0..70]]
   ActionDir turn_signals_front{ActionDir::NONE}; // TAL types created automatically for enum, bool, etc.
   static constexpr int i_FREE_LANE = -1;         // Constants are replaced directly in the code.
};

struct Agent {
   int v{}; // #vfm-tal[[0..34]]
   int a{}; // #vfm-tal[[-8..6]]
   bool has_close_vehicle_on_left_left_lane{};
   bool has_close_vehicle_on_right_right_lane{};
   bool flCond_full{}; // Lane change towards the "fast lane" (fl), i.e., left lane in our case.
   bool slCond_full{}; // Lane change towards the "slow lane" (sl), i.e., right lane in our case.
   std::array<Gap, 3> gaps;
};

void plan( // Entrance function for Model Checking, indicated by the "vfm-gencode" tag below.
   Agent& agent // #vfm-aka[[ literal(ego) ]] // Map variable name "agent" to "ego" in EnvModel
   ) {
   // #vfm-gencode-begin[[ condition=false ]]
   static constexpr int MIN_DIST = 15;
   agent.flCond_full = false; // Fast lane condition.                 // K2
   agent.slCond_full = false; // Slow lane condition.                 // K2
                                                                      // K2
   if (agent.gaps[ActionDir::CENTER].s_dist_front < MIN_DIST          // K2
       && agent.gaps[ActionDir::LEFT].s_dist_rear > MIN_DIST          // K2
       && agent.gaps[ActionDir::LEFT].s_dist_front > MIN_DIST         // K2
       && agent.v < agent.gaps[ActionDir::LEFT].v_front               // K2
       && agent.v > agent.gaps[ActionDir::CENTER].v_front) {          // K2
      agent.flCond_full = true;                                       // K2
   } else if (agent.gaps[ActionDir::CENTER].s_dist_front < MIN_DIST   // K2
              && agent.gaps[ActionDir::RIGHT].s_dist_front > MIN_DIST // K2
              && agent.gaps[ActionDir::RIGHT].s_dist_rear > MIN_DIST  // K2
              && agent.v < agent.gaps[ActionDir::RIGHT].v_front) {    // K2
      agent.slCond_full = true;                                       // K2
   }                                                                  // K2

   bool ego_pressured_from_ahead_on_left_lane = agent.gaps[ActionDir::LEFT].turn_signals_front ==
         ActionDir::RIGHT;
   bool ego_pressured_from_ahead_on_right_lane = agent.gaps[ActionDir::RIGHT].turn_signals_front ==
         ActionDir::LEFT;
   int ego_following_dist = std::max(2 * agent.v, 5/*0*/); // #vfm-tal[[5..78]]
   int allowed_ego_a_front_center = std::max(std::min(agent.gaps[ActionDir::CENTER].s_dist_front +
         agent.gaps[ActionDir::CENTER].v_front + agent.gaps[ActionDir::CENTER].a_front - agent.v -
         ego_following_dist, 2), -8);
   int allowed_ego_a_front_right = std::max(std::min(agent.gaps[ActionDir::RIGHT].s_dist_front +
         agent.gaps[ActionDir::RIGHT].v_front + agent.gaps[ActionDir::RIGHT].a_front - agent.v -
         ego_following_dist, 2), -8);
   int allowed_ego_a_front_left = std::max(std::min(agent.gaps[ActionDir::LEFT].s_dist_front +
         agent.gaps[ActionDir::LEFT].v_front + agent.gaps[ActionDir::LEFT].a_front - agent.v -
         ego_following_dist, 2), -8);
   int acceleration = std::min(allowed_ego_a_front_center, 2);

   if (agent.gaps[ActionDir::CENTER].s_dist_front < 10
        || (agent.gaps[ActionDir::LEFT].s_dist_front < 10 && ego_pressured_from_ahead_on_left_lane)
        || (agent.gaps[ActionDir::RIGHT].s_dist_front < 10 && ego_pressured_from_ahead_on_right_lane)) {
        acceleration = std::max(-agent.v, -8);
   } else {
        if (ego_pressured_from_ahead_on_right_lane) {
           acceleration = std::min(acceleration, allowed_ego_a_front_right);
        }
        if (ego_pressured_from_ahead_on_left_lane) {
           acceleration = std::min(acceleration, allowed_ego_a_front_left);
        }
   }

   agent.a = acceleration;
   // #vfm-gencode-end
}
// #vfm-end
```

Algorithm 2: Transition system in SMV. The SMV code indicates how the BP can be integrated with an EM, both given in SMV (`planner.k2.smv` and `envmodel.smv`). (Square-bracket array notation `...[i]` used for readability, cf. Sec. A.1.)

```
#include "planner.k2.smv" -- Include SMV version of BP, generated from C++.
#include "envmodel.smv"   -- Include EM.

MODULE Globals
VAR
"loc" : boolean;

MODULE main
VAR
  globals : Globals;
  env : EnvModel;
  planner : "plan"(globals,  -- 'Call' to plan function of BP.
                  env.ego.gaps[0].s_dist_front,
                  env.ego.gaps[0].s_dist_rear,
                  env.ego.gaps[0].v_front,
                  env.ego.gaps[1].s_dist_front,
                  env.ego.gaps[1].v_front,
                  env.ego.gaps[2].s_dist_front,
                  env.ego.gaps[2].s_dist_rear,
                  env.ego.gaps[2].v_front,
                  env.ego.v);

INIT !env.ego.flCond_full; -- Fast lane condition, set initially to FALSE.
INIT !env.ego.slCond_full; -- Slow lane condition, set initially to FALSE.

TRANS env.ego.flCond_full = planner."agent.flCond_full"; -- Take over BP output.
TRANS env.ego.slCond_full = planner."agent.slCond_full"; -- Take over BP output.

INVARSPEC !env.blamable_crash;  -- Example INVAR specification to check.
-- LTLSPEC G !env.blamable_crash;  -- (Same specification expressed as LTL.)
```

Algorithm 3: Program in K2. The K2 code corresponds to the logic excerpt of the BP code tagged "K2" in Alg. 1. It is automatically derived from C++, and further translated into SMV by our toolchain (`planner.cpp` ⇒ `planner.k2` ⇒ `planner.k2.smv`). (Square-bracket array notation `...[i]` used for readability, cf. Sec. A.1.)

```
(entry plan) ;; Function 'plan' is the main function.
(globals (! (var loc bool) :location-var true))
(function plan ((! (var |agent.gaps[0].s_dist_front| int) :input-var true)
                (! (var |agent.gaps[0].s_dist_rear| int) :input-var true)
                (! (var |agent.gaps[0].v_front| int) :input-var true)
                (! (var |agent.gaps[1].s_dist_front| int) :input-var true)
                (! (var |agent.gaps[1].v_front| int) :input-var true)
                (! (var |agent.gaps[2].s_dist_front| int) :input-var true)
                (! (var |agent.gaps[2].s_dist_rear| int) :input-var true)
                (! (var |agent.gaps[2].v_front| int) :input-var true)
                (! (var |agent.v| int) :input-var true))
(return (var |agent.flCond_full| bool)
        (var |agent.slCond_full| bool)) ;; Interface of 'plan' from C++.
(locals)

(seq ;; IF/ELSE logic from C++ expressed with K2 language elements.
(assign |agent.flCond_full| false)
(assign |agent.slCond_full| false)
(condjump (not (and (and (and (and (lt |agent.gaps[1].s_dist_front| (const
     15 int)) (gt |agent.gaps[0].s_dist_rear| (const 15 int))) (gt
     |agent.gaps[0].s_dist_front| (const 15 int))) (lt |agent.v|
     |agent.gaps[0].v_front|)) (gt |agent.v| |agent.gaps[1].v_front|)))
     (label else3))
(assign |agent.flCond_full| true)
(jump (label endif3))
(label else3)
(condjump (not (and (and (and (lt |agent.gaps[1].s_dist_front| (const 15
     int)) (gt |agent.gaps[2].s_dist_front| (const 15 int))) (gt
     |agent.gaps[2].s_dist_rear| (const 15 int))) (lt |agent.v|
     |agent.gaps[2].v_front|))) (label else4))
(assign |agent.slCond_full| true)
(jump (label endif4))
(label else4)
(label endif4)
(label endif3)))
```