# Translation of UML Statecharts to UPPAAL Automata for Verification of Real-time Systems

Daniil A. Zorin
Department of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, Russia
juan@lvk.cs.msu.su

Vladislav V. Podymov
Department of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, Russia
vvpodymov@gmail.com

*Abstract* — **In this paper we present a tool to transform UML statecharts to UPPAAL automata. The tool allows one to check temporal properties against statecharts modeling a real-time system. We give the constraints on statecharts, the tool description, and the results of testing it on a well-known traffic control example.**

***Keywords — verification, UML, UPPAAL, modeling, real-time systems***

## I. Introduction

Usually verification tools work with models written in specialized languages intended for convenient application of verification algorithms. On the other hand, during the design stage systems are often modeled with universal modeling languages (such as UML) or industry-specific modeling languages. UML statechart diagrams are an example of universal models describing the behavior of systems communicating with the environment via shared memory and message passing. Real-time systems are often modeled with such diagrams. Since the cost of correcting an error increases over the course of system development, verifying the properties of the system as early as possible one improves its quality and simplifies its development.

In this paper we present a tool for converting UML statechart diagrams to timed automata used in the UPPAAL verification system [1, 2]. In section 2 we define the syntax of expressions we use in UML diagrams. The algorithm is discussed in section 3. Experimental results obtained with the algorithm are given in section 4.

## II. UML Statecharts

Unified Modeling Language (UML) is used to design a wide range of systems implemented in various languages and in different environments. Therefore, the authors of the standard of UML deliberately avoid defining syntax and semantic of the language completely [5, ch. 13]. The language defines a metamodel comprised of syntactical constraints on all models in UML notation. Generally it is only possible to say whether the model is syntactically correct. The behavior of a correct model might be undetermined in some cases: guards, actions and triggers can be defined in a natural language which tolerates different interpretations.

The authors of the language suggest creating a separate profile for every class of systems without changing the general notation. However, in the case of statechart diagrams, creating the profile does not solve interpretation problems. To prove the properties formally it is necessary to define a strict syntax and semantic of all used primitives of statecharts. In this study, additional constraints on the structure of the diagrams and the syntax of expressions are imposed, thus the ambiguity is avoided.

Simple states are the same as in the standard UML metamodel. There are two types of composite states: sequential and parallel. Automata residing in a parallel state are executed simultaneously. Composite states have special entry and exit states.

Some states are marked with logical formulae called invariants; a system can reside in such state only while its invariant is satisfied.

Each transition between states may be provided with a guard, an action, and a synchronization. Guards express requirements that must be satisfied to enable the transition. Actions are the operations performed after the transition is fired.

The syntax of guards, invariants and actions is similar to the syntax of the C language. There are three types of variables: an integer type over a certain range (e.g., int [4..9] x = 5;), the boolean type (e.g., bool b = false;), and the clock type (e.g., clock c;). All variables must be defined in the comments section of the UML model. Expressions admitted in guards include all types of comparison as well as logical NOT, AND and OR operations. Actions may contain assignment statements including complex arithmetic expressions and the C-style ternary operator ‘? :’. Invariants have the same syntax as guards do, though the expression must be marked with the keyword ‘assume()’.

There are two additional expressions in the syntax. The boolean expression 'in(S)' borrowed from STATEMATE language [3] denotes that the state S is active in the system. The operation of random assignment, written as ‘x=random();’ non-deterministically gives a value to an integer or boolean variable admitted by the type.

The syntax and the meaning of macros are similar to the ones in C language. They are defined in the comments section along with the variables. The macro '#define X Y' replaces all occurrences of X with Y before other stages of the translation.

The examples of expressions can be found in the Figure 9.

The operation of sending a signal is identical to the hardware-like message broadcast [3]. Every signal must be defined in the model. When signal S is sent by a transition (denoted by the synchronization section), the automaton marks signal S as sent, and on the next step all the automata that can activate a transition with the receiving of signal S (written as '!!S') must do that. If none of the automata can receive the signal, it is considered lost. For instance, on figure 10 the system moves from state AHome to state AToGreen only at receiving a signal AtoG.

## III. Translation of UML statecharts to UPPAAL automata

The UML to UPPAAL translator works with UML statecharts in the widespread XMI format. When a file is parsed and an internal representation is constructed, the translation is performed in two phases. First, the statechart is transformed to the intermediate form – an hierarchical timed automaton (HTA) [4] – and then this automaton is translated to a network of timed automata (NTA) according to the algorithm similar to the one introduced in [4].

Since the structure of statecharts differs significantly from the structure of hierarchical timed automata, an additional step of transformation of UML statecharts should be carried out before translating them to UPPAAL.

Firstly, during the parsing of UML, the expressions that do not belong to UPPAAL model language are translated. All macro substitutions take place before parsing the guards and actions. The 'in(S)' expression in guards is replaced by checking the value of a special flag variable which is unique for each 'in(S)' statement.

Further, all references to automata are replaced by their unique copies. If one automaton is nested into another one, it is inserted as well. Name collision on this step is avoided: if the names of two states in two nested automata coincide, then one of the states is renamed, and if two variables with the same name are declared in different scopes (e.g. in two automata referenced in the third one), then one of them is renamed. As a result a single hierarchical UML statechart is formed.

The next step is to modify composite states (figure 2-3). In HTA, only transitions between simple states, entry and exit states are allowed, so it is necessary to change the arcs which start or end in composite states to match them with the corresponding entries and exits. Adding several new entries or exits might be necessary. In HTA transitions into a composite state are allowed if they end in its entry state, similarly, transition out of a composite state into its parent is possible if it starts in an exit state. All other transitions must begin and end inside of the same composite state, i.e., the source and target states remain in the same composite state. However, in UML statecharts it is possible to perform transitions to a deeply nested state; hence it is important to add all exits and entries in between.

Finally, guards, actions, and synchronizations should not be present on transitions ending in exit states according to HTA definition. In such cases, a new state, like tmp in the Figure 2, is added and the guards, actions and synchronizations are assigned to the transition ending in the new state.

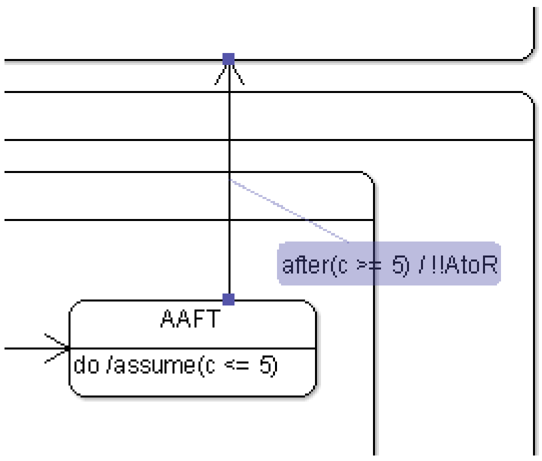

Figure 1. Correction of composite states: before

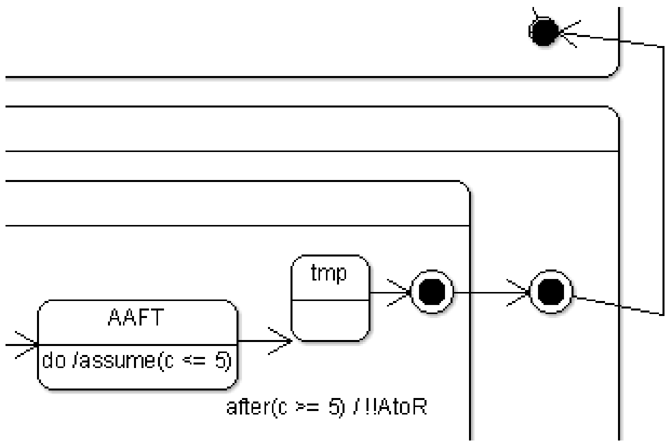

Figure 2. Correction of composite states: after

When HTA is obtained, it is translated into NTA used in UPPAAL.

NTA consists of processes, variables, channels and clocks. A process is a certain timed automata which has finite sets of locations and transitions.

Some locations are marked with invariants, and some transitions are supplied with guards, actions, and synchronizations. Invariants, guards, actions, and synchronizations are similar to those in HTA.

Three kinds of locations are possible: ordinary, urgent, and committed. When an urgent location is active in NTA, no time can advance, and if the location can be deactivated, it is left at once. Committed locations are similar to urgent locations, but they have the highest priority in deactivation.

Each channel has its own type, either broadcast or handshake. Broadcast channels are similar to those in HTA. Handshake channel is used to synchronize the execution of exactly two transitions in NTA.

The translation HTA to NTA is as follows.

Before state translation, variables, channels and clocks are copied from HTA directly to NTA. According to translation algorithm, auxiliary variables and channels are added. Some of them are mentioned below.

Every composite state S in HTA corresponds to a process P(S) in NTA. Every such a process has an initial location ‘idle’ which corresponds to inactivity of a composite state.

Consider a parallel composite state S in HTA. A special location ‘active’ is created in P(S). The ‘active’ location can be reached from the ‘idle’ location by performing a sequence of transitions via committed locations ‘start(X)’, one for each composite state X nested in S. The first transition in the sequence carries a synchronization ‘activate(S)?’ that activates P(S). Other transitions in the sequence carry synchronizations ‘activate(X)!’ for every nested state X. Also there is exactly one transition from the state ‘active’ to the state ‘idle’ that carries a synchronization ‘deactivate(S)?’ deactivating P(S).

When P(S) is activated, the whole sequence of transitions is executed with no time advancing, every nested state is activated, and then P(S) reaches the ‘active’ location which corresponds to activity of all states nested in S.

Consider a sequential composite state S in HTA. A process P(S) includes locations ‘active(X)’ for every state X nested in S as well as committed locations ‘start(X)’ for every composite state nested in S. Locations ‘start(X)’ and ‘active(X)’ are connected via a transition decorated with a synchronization ‘activate(X)!’. The ‘idle’ location is connected with either a location ‘start(X)’ in the case of a composite state X or with a location ‘active(X)’ in the case of a basic state X via transition with synchronization ‘activate(S)?’.

When P(S) is activated, it activates exactly one of its nested states and reaches one of ‘active(X)’ locations which correspond to the activity of X.

To deactivate a state X nested in S, the process P(S) uses a set of deactivation sequences of committed locations. Transitions in each sequence carry synchronizations ‘deactivate(Y)!’ for every composite state Y nested at any level in X. Thereby when a deactivation sequence is executed, all inner states which can be deactivated simultaneously in HTA are deactivated in NTA. If S has to be deactivated as well, the final location of the sequence is connected to the ‘idle’ location. Otherwise it is connected to one of ‘start(X)’ or ‘active(X)’ locations.

To initialize the NTA defined above, an additional process ‘Kickoff’ is created. This process is a sequence of committed locations which ends with an ordinary location. Transitions in this process carry special synchronizations ‘init(X)!’ for every initial state X of HTA. Special initial transitions are also added into other processes to reach a correct initial state.

## IV. Experimental Results

To be certain that the implementation of our translation algorithm is correct and well suited for composition with UPPAAL we tested it on several case studies. The simple examples were used to make sure that the output of the algorithm satisfies the expectations and to check the behavior on various sample cases. Some more complex tests were aimed to simulate the whole process of verification of a system defined by a UML statechart diagram. Below we present the results of our experiments with the model of traffic lights control system described in [4].

### A. *Simple tests*

An example of a simple test is given on figures 4-5.

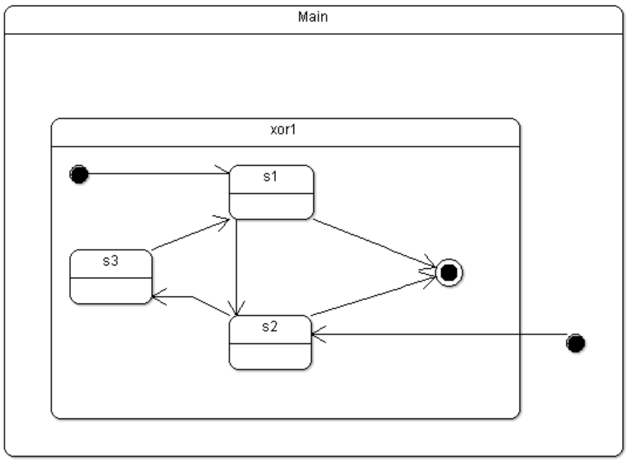

Figure 3. Example 1: UML

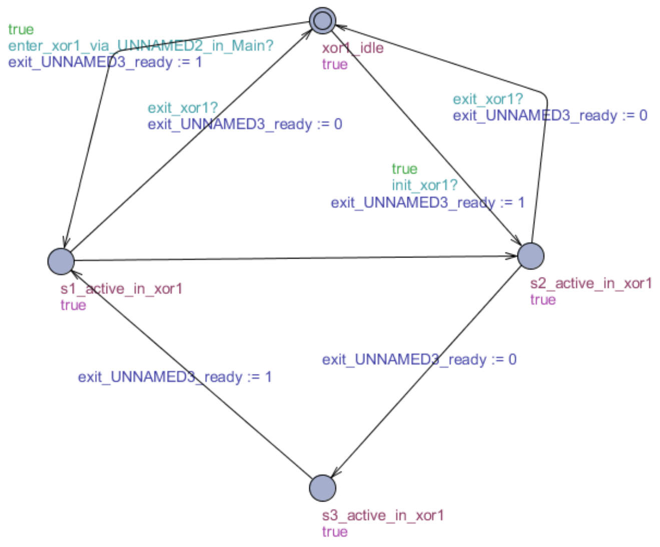

Figure 4. Example 1: UPPAAL

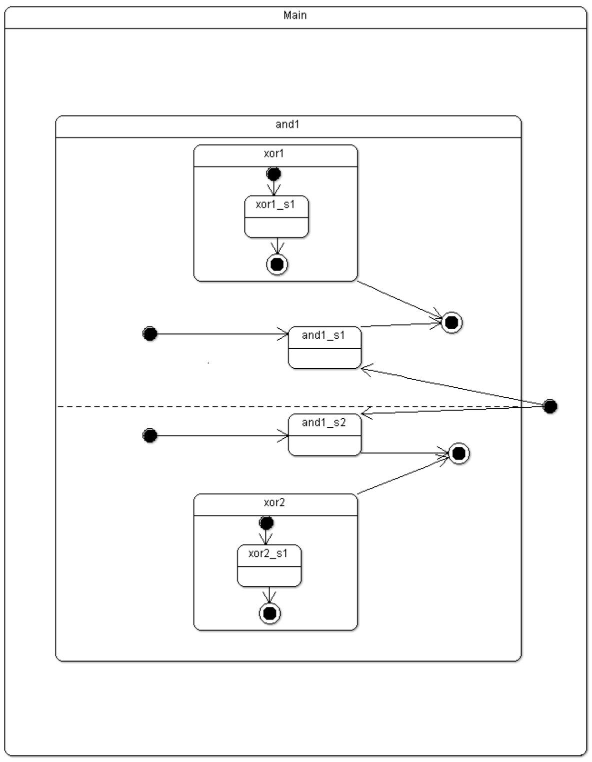

Figure 5. Example 2: UML

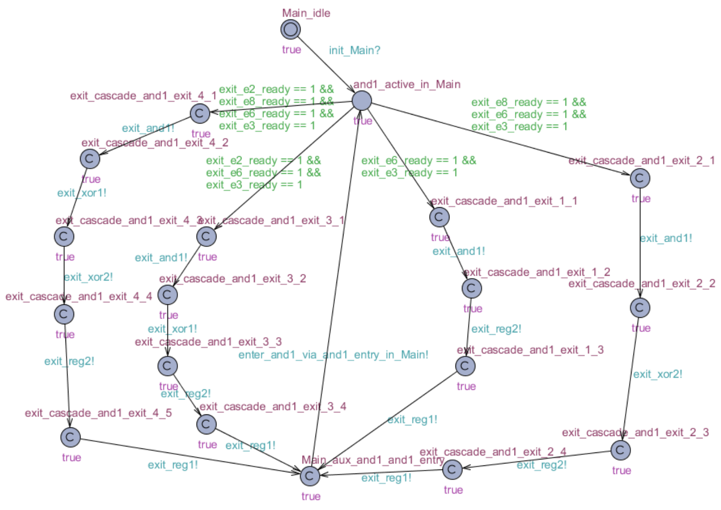
Main_idle
true
init_Main?
and1_active_in_Main
exit_cascade_and1_exit_4_1
exit_e2_ready == 1 &&
exit_e6_ready == 1 &&
exit_e3_ready == 1
exit_e8_ready == 1 &&
exit_e6_ready == 1 &&
exit_e3_ready == 1
exit_and1!
exit_cascade_and1_exit_4_2
exit_xor1!
exit_cascade_and1_exit_4_3
exit_cascade_and1_exit_3_1
exit_cascade_and1_exit_1_1
exit_cascade_and1_exit_2_1
exit_xor2!
exit_cascade_and1_exit_4_4
exit_cascade_and1_exit_3_2
exit_cascade_and1_exit_1_2
exit_cascade_and1_exit_2_2
exit_reg2!
exit_cascade_and1_exit_3_3
enter_and1_via_and1_entry_in_Main!
exit_cascade_and1_exit_1_3
exit_cascade_and1_exit_4_5
exit_cascade_and1_exit_3_4
exit_reg1!
exit_cascade_and1_exit_2_3
exit_cascade_and1_exit_2_4
Main_aux_and1_and1_entry
true

Figure 6. Example 2: UPPAAL

## *B. Traffic lights example*

The traffic lights control system consists of two traffic lights on a crossroad. The lights are controlled by a processor supplied with some sensors. Lights on the street and on the avenue change colors customary to let cars pass by in both directions. Further, in the case an ambulance car arrives from any direction, the lights must turn to green on that direction in order to let the ambulance pass as soon as possible.

The UML diagrams for this system are shown in the Figures 10-11. The first diagram contains state loops for the lights and the ambulance and a reference to the diagram of the light controller. The lights are changed in the usual order (green to yellow to red) according to the signals of the light controller. The ambulance appears non-deterministically and passes through the street crossing.

The light controller normally sends signals to the lights to switch their colors in order. When the ambulance appears, the system exits the normal cycle and enters the AmbulanceArriving composite state where the light colors are changed arbitrarily in order to turn the light on the street where the ambulance is waiting green.

In [4] the authors constructed a UPPAAL model for this system manually to verify its properties. We used our translator and obtained the model automatically.

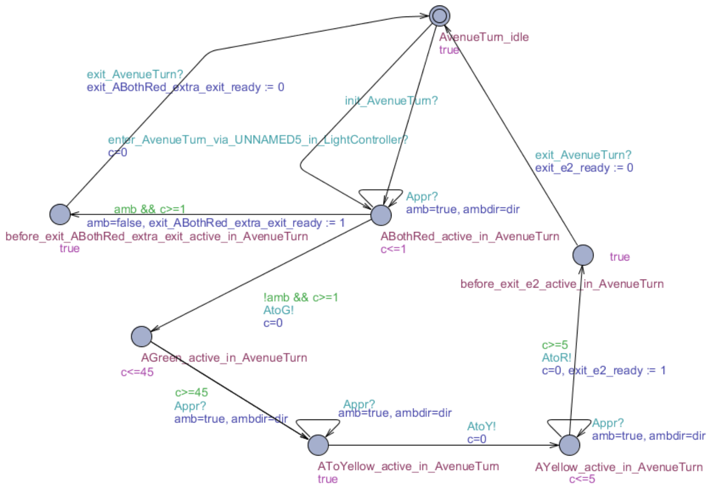
AvenueTurn_idle
true
exit_AvenueTurn?
exit_ABothRed_extra_exit_ready := 0
init_AvenueTurn?
enter_AvenueTurn_via_UNNAMED5_in_LightController?
c=0
exit_AvenueTurn?
exit_e2_ready := 0
amb && c>=1
amb=false, exit_ABothRed_extra_exit_ready := 1
Appr?
amb=true, ambdir=dir
before_exit_ABothRed_extra_exit_active_in_AvenueTurn
true
ABothRed_active_in_AvenueTurn
c<=1
true
before_exit_e2_active_in_AvenueTurn
!amb && c>=1
AtoG!
c=0
AGreen_active_in_AvenueTurn
c<=45
c>=5
AtoR!
c=0, exit_e2_ready := 1
c>=45
Appr?
amb=true, ambdir=dir
AtoY!
c=0
ATоYellow_active_in_AvenueTurn
true
AYellow_active_in_AvenueTurn
c<=5

Figure 7. UPPAAL diagram for AvenueTurn composite state

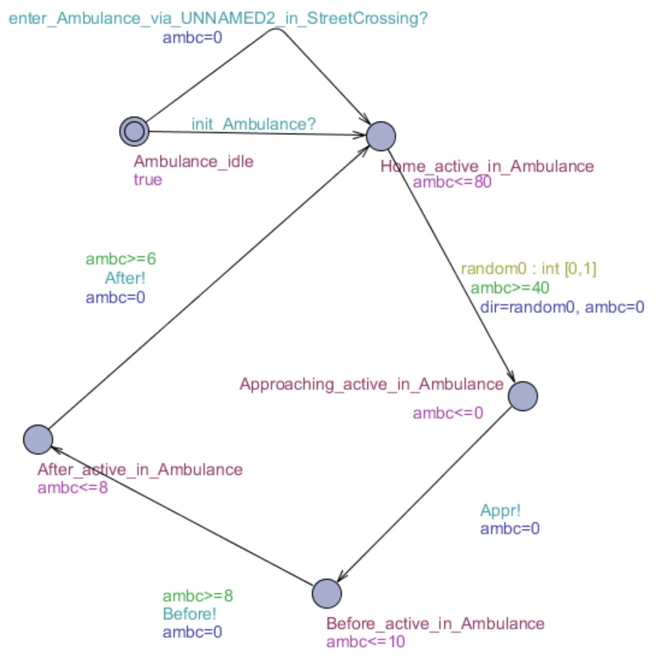

Figure 8. UPPAAL diagram for Ambulance behavior

Some of the UPPAAL automata are shown on figures 7-8.

The following properties were tested.

*A[]! deadlock*

This property guarantees the absence of deadlocks.

*A[]! (stg==1 || sty==1) imply avr==1*

*A[]! (avg==1 || avy==1) imply str==1*

The lights are synchronized: if the avenue light is green or yellow, the street light must be red and vice versa.

*E<> stg==1 && avg==1*

This property means that there exists a trace where both lights are green at the same time and it was proved to be false. At the same time the seemingly contrary property

*A[] (stg==1 || avg==1)*

is not fulfilled also, because there can be a situation where one light is red and the other one is yellow.

*Ambulance_process_proc.Approaching_active_in_Ambulance* -->

*Ambulance_process_proc.Home_active_in_Ambulance*

Home state for the ambulance car is reachable from the Approaching state, which basically means that the ambulance will always eventually pass the crossing.

## CONCLUSIONS

Experiments with our tool testify that translation of UML statecharts to UPPAAL timed automata is possible. We reproduced the results that were obtained manually in [4] with our automatic translation and showed that the tool is applicable to models of relatively simple real-time systems with parallel interacting processes. Further work includes formal proof of the correctness of the algorithm based on [3] and testing the tool on practical examples of real-time systems.

## REFERENCES

[1] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. Int. Journal on Software Tools for Technology Transfer, 1(1–2):134–152, October 1997.

[2] Alexandre David, Gerd Behrmann, Kim G. Larsen, and Wang Yi. A tool architecture for the next generation of UPPAAL. In 10th Anniversary Colloquium. Formal Methods at the Cross Roads: From Panacea to Foundational Support, LNCS, 2003.

[3] David, M. Moller Oliver, Wang Yi. Verification of UML Statechart with Real-Time Extensions / Uppsala: Department of Information Technology, Uppsala University. IT Technical Report 2003-009, 2003.

[4] A. Furfaro, L. Nigro. A development methodology for embedded systems based on RT-DEVS, Innovations in Systems and Software Engineering, vol 5, P. 117-127, June 2009.

[5] Grady Booch, Ivar Jacobson & Jim Rumbaugh. OMG Unified Modeling Language Specification. Addison Wesley, 1997

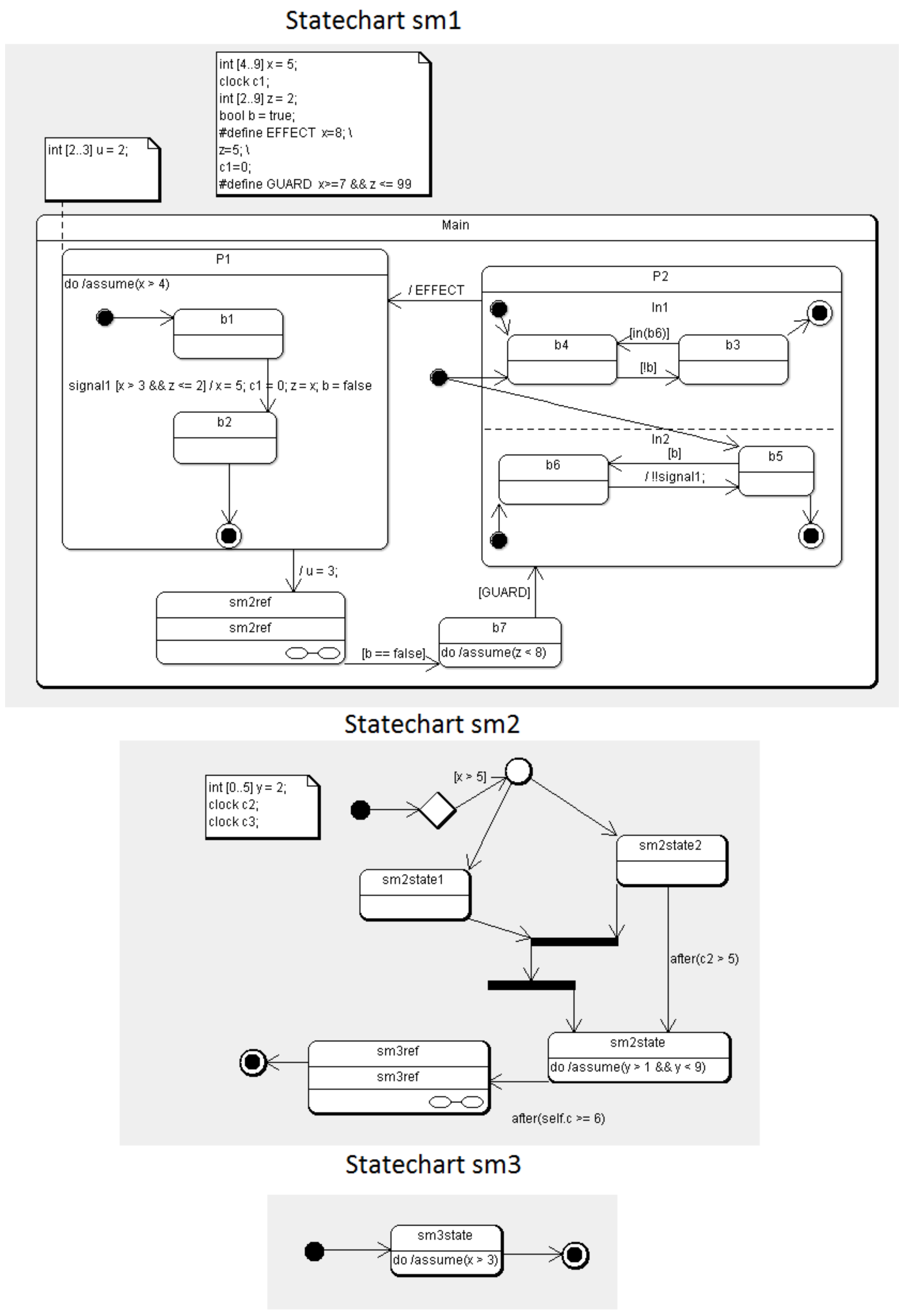

Figure 9. Example of a UML diagram containing all syntactic features

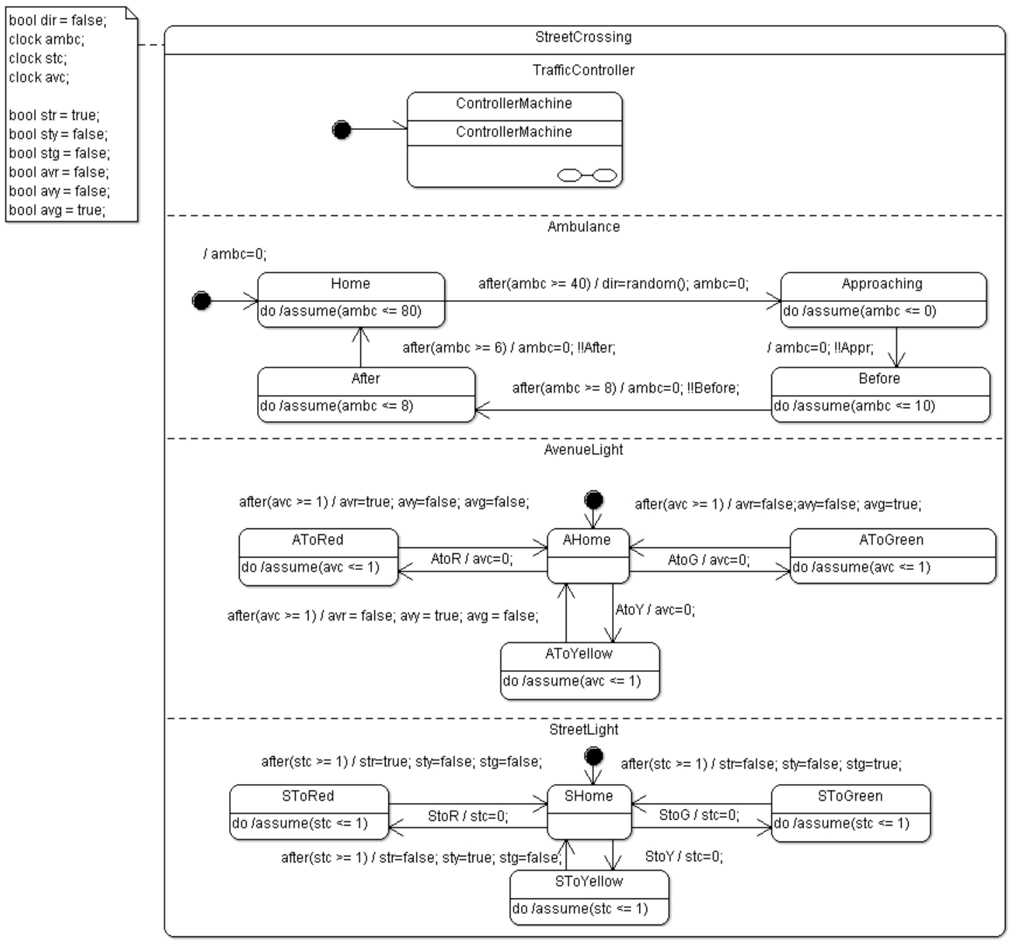

Figure 10. Traffic lights UML diagram

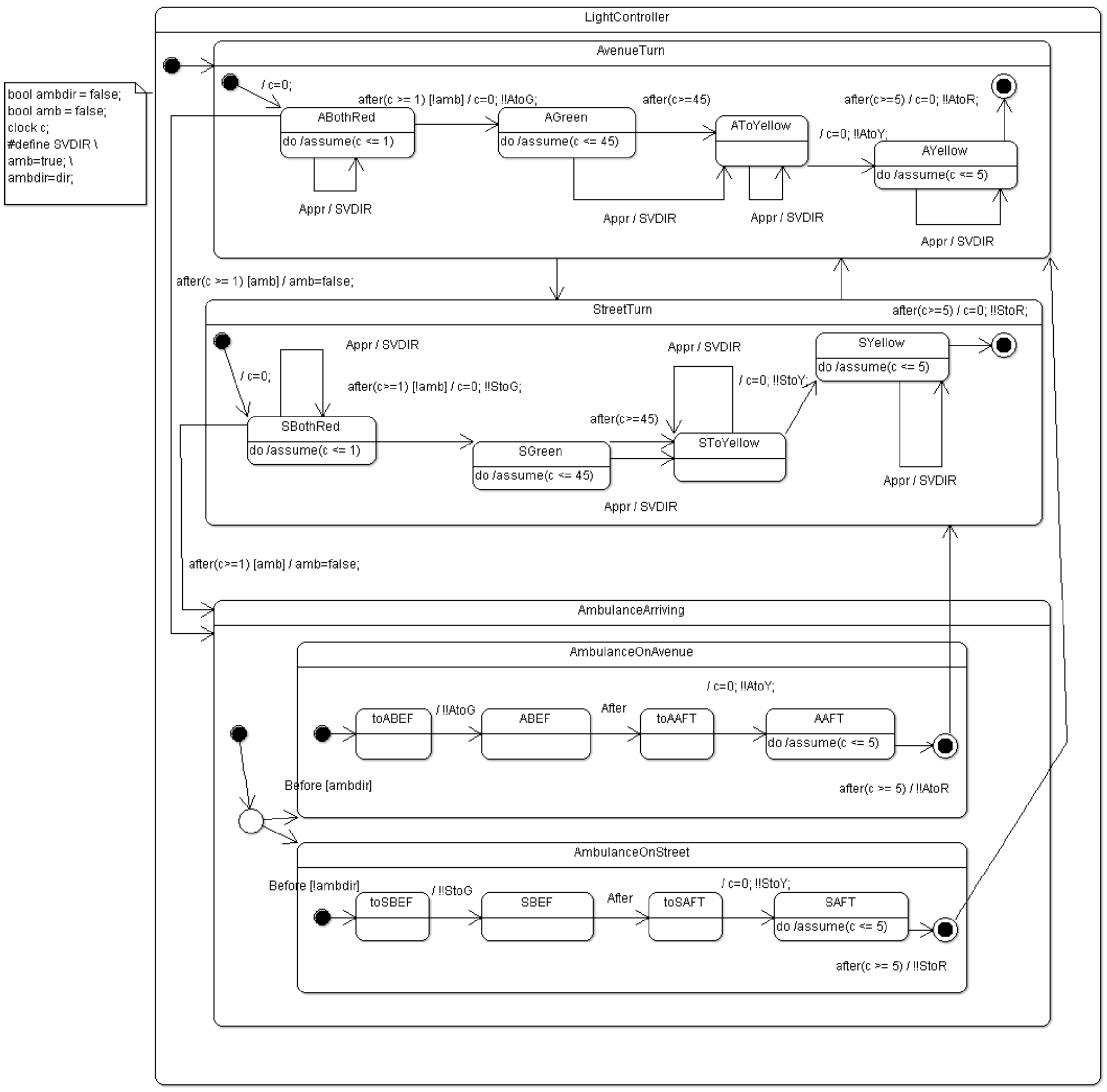

Figure 11. Traffic lights UML diagram (2)