Behavioural Verification in Embedded Software, from Model to Source Code

Anthony Fernandes Pires & Thomas Polacsek & Virginie Wiels & Stéphane Duprat*

Abstract

To reduce the verification costs and to be more confident on software, static program analysis offers ways to prove properties on source code. Unfortunately, these techniques are difficult to apprehend and to use for non-specialists. Modelling allows users to specify some aspects of software in an easy way. More precisely, in embedded software, state machine models are frequently used for behavioural design. The aim of this paper is to bridge the gap between model and code by offering automatic generation of annotations from model to source code. These annotations are then verified by static analysis in order to ensure that the code behaviour conforms to the model-based design. The models we consider are UML state machines with a formal non-ambiguous semantics, the annotation generation and verification is implemented in a tool and applied to a case study.

1. Introduction

Aeronautical software development, and more specifically software for safety critical applications, is submitted to stringent constraints. DO-178C[1] (certification standard for aeronautical software) specifies development and verification objectives. Identified verification means are reviews, analyses and test. One of its supplements, DO-333[2], is dedicated to the use of formal methods. Formal methods are mathematical techniques which allow performing rigorous verification tasks during software development. Formal methods are already applied in industry [19].

In an industrial context, at Atos, we notice that the cost of verification activities for embedded software development can sometimes reach 60% of the project workload. This is not a new problem, Hoare [15] was already reporting that over half of software development time was dedicated to program testing.

Furthermore, in addition to the increasing complexity of embedded systems, today software is not developed by a single company but by a set of stakeholders. These stakeholders have a common purpose and share some resources and knowledge. In this context, it can be difficult to communicate between all the different stakeholders. It is essential to offer a way for muti-cultural teams to share, to discuss and to work with an unambigous formalism. Model Driven Engineering (MDE) allows us to deal with these difficulties while ensuring the expected level of quality. It suggests using models all along the development lifecycle; models can be used, for instance, for documentation generation, design specification, simulation or code generation.

In this paper, we present an MDE approach to combine the advantages of model-based design and the efficiency of formal methods dedicated to code verification. More specifically, we give a process to support the design, development and verification of the implementation of software for the management of avionic components.

Many modelling languages have been defined through past decades. The UML[3] standard is one of them. UML is widespread and it is currently used in Atos development teams. The current

*Authors version, Behavioural verification in embedded software, from model to source code. In International Conference on Model Driven Engineering Languages and Systems, pp. 320-335. Springer, Berlin, Heidelberg, 2013.

[1] DO-178C *Software considerations in airborne systems and equipment certification*

[2] DO-333 *Formal Methods Supplement to DO-178C and DO-278A*

[3] Unified Modeling Language `www.uml.org`

UML semantics is semi-formal as it is partially expressed in natural language. However, the UML standard has known a significant evolution in its description since version 1.x. The Precise UML group [4] contributed to this evolution. It aimed at investigating a precise semantics for UML. In [11], authors explain that the lack of precise semantics results in, among other, difficulties to rigorously establish the consistency of a model and its implementation.We propose to exploit the UML standard to model the design of embedded software. The design represents all the information needed to directly implement the sofware.

In our approach, this implementation could be done by automatic generation from models or by humans. Both solutions fit and, in this paper, we simply define an implementation pattern for our UML model. We need this pattern to manage an automatic verification task.

The main contribution of this paper is automatic verification of a C code stemming from an UML state machine. We want to prove that a source code implements and only implements its model based design. This verification is done using static analysis. Static analysis allows the detection of bugs and the verification of properties on a program without executing it. It enables effective identification of software defects and allows reducing verification costs. We propose to automatically generate annotations from the model into the code implementation. These annotations represent the behavioural properties of the model. They will be automatically verified by a static analysis tool.

The paper is structured as follows. Section 2 gives the definition of our language, subset of UML state machines, and its formal semantics. Section 3 describes our process, give an implementation pattern for our state machines and explain the annotations generation. Section 4 presents a prototype that implements our method. Section 5 reviews existing related work. Lastly, Section 6 concludes the paper and outlines perspectives to this work.

2. State machine modelling in embedded software context

2.1. Modelling Language

In [10] we define a UML subset dedicated to embedded software specification and already used for industrial purpose. In this subset, we use UML state machines to represent the behavioural specification of software components. We limit the scope of elements and we define patterns for specific use, without adding new concepts. These state machines are meant to be driven by a clock and to do a certain number of actions at each clock *tick*.

Here, we use a limited subset of this language. Our state machines are composed of simple states, which can contain actions defined in their *entry behaviour*. In UML, an action defined in the *entry behaviour* is executed to completion at the entry into the state. Hierarchical states and parallelism behavior are disallowed for the moment. We have transitions between states. A transition is composed of a trigger, to manage event received by the state machine, and a guard, representing the condition to fire the transition. The trigger can be defined by only two events: the tick event and the completion event. The tick represents our clock tick and the beginning of a new cycle. The completion event is a special event defined in UML, it represents the default event of the triggers which is automatically generated at the end of all the actions of a state or at the entry of state if no actions are defined. The guard is represented by a boolean expression expressed in the OCL standard language[5]. We do not allow the definition of actions on the transitions, i.e. effects of the transitions. We authorise a unique pseudostate by state machine: the initial state. These are the only UML elements used to model our cyclic state machine.

[4] www.cs.york.ac.uk/puml/

[5] http://www.omg.org/spec/OCL/

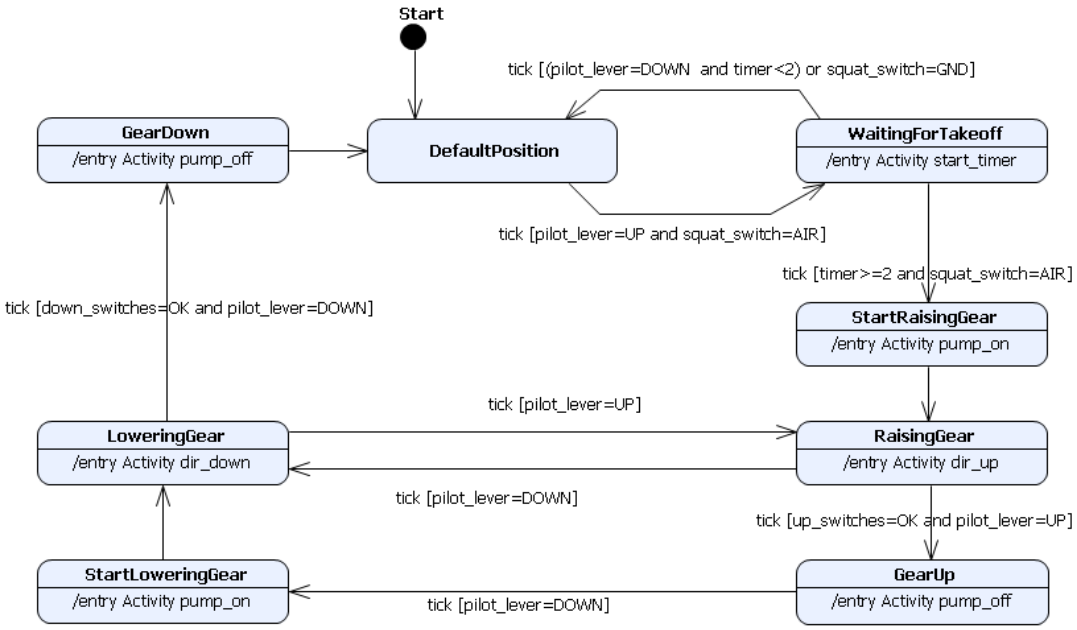

Figure 1: *LandingGear state machine*

We add two constraints to these state machines. In the first constraint, we consider that all the actions executed in one cycle end before the end of the cycle. As we are not interested in time properties, we accept the synchrony hypothesis defined in [4]. It considers that every reaction of the system is instantaneous. In the second constraint, we consider that the state machine must be deterministic. We do not authorise conflicting transitions.

Consider the example of Figure 1. It is based on the example described in [13] and it illustrates the behaviour of the software controling the landing gear of an UAV (Unmanned Aerial Vehicle). The landing gear is composed of three gears: a nose gear, a left gear and a right gear. Each of these gear has an up switch and a down switch, namely `up_switches` or `down_switches` in our example. Each switch is closed when the gear is respectively up or down. An additional switch on the aircraft, named the `squat_switch`, indicates if the weight of the plane is on the nose or not. If the weight is on the nose, corresponding to `squat_switch=GRD` in Figure 1, it means that the plane is still on ground; if not, `squat_switch=AIR`, it means that the plane is in the air. The raising or lowering of the landing gear is managed by an electrically driven hydraulic pump. This pump supplies pressure to the gear actuators. The pressure increases or decreases depending on a computer-driven valve. When the pilot wants to raise the gears, He raises a lever. When the lever is up, `pilot_lever=UP`, the pump is activated and the pressure level is set, corresponding respectively to the actions `Activity pump_on` and `Activity dir_up`. When the pilot wants to lower the gears, He lowers a lever, `pilot_lever=DOWN` the pump is also activated, `Activity pump_on`, and the pressure level is set, `Activity dir_down`.

Starting in the default position, if we look for instance at the takeoff phase, the pilot raises the lever and the aircraft needs to be airborne for two seconds (`timer>2 and squat_switch=AIR`) before starting raising gears. This allows ensuring the aircraft does not touch the ground during takeoff. After two seconds, the pump is activated and gears are raised. When gears are closed, the pump is deactivated and the gears are in the up position.

2.2. Semantics

We propose to formalise the semantics of our subset, in compliance with the UML semantic basis. To understand it, the reader needs to be familiar with some UML specific concepts.

In UML, state machines behaviour is managed by event processing. Each state machine has an event pool to store events, its politic of dequeuing must be defined by the user. The concept of event processing is called Run-to-Completion which limits the processing of events to one at a time. When an event is taken from the pool, if it enables a transition i.e. it fires a transition according to the correctness of its guard, it is consumed. If not, the event is simply discarded. The processing of a single event is called a Run-to-completion step. It represents the passage between two stable state configurations of a state machine. A state machine is in a stable state configuration when it is in a state where all the state actions have been completed: if a transition is fired at the beginning of the run-to-completion step, this step ends when all the actions of the targeted state are completed.

In our state machine, there are only two types of events, the *tick* event which is an external event periodically given by the environment and the completion event which is an internal event. The completion event is a very particular event defined in UML. It is automatically generated at the end of all the actions of a state and it has priority over all events existing in the event pool. To formally describe our semantics in a very simple way, we decide to make an abstraction of the concept of event and only use the concept of run-to-completion step.

We define S the set of all states of the state machine, VAR the set of variables accessible by the state machine. $s_0 \in S$ is the initial state of the state machine.

We define v as the variable assignment that associates an element of the domain of discourse with each variable of VAR. We define v_0 the variable assignment for s_0 and V the set of all variable assignments.

We define $T : S \times V \to S \cup \{\emptyset\}$, the transition function. For each state, T returns a new state according to the transition guards. A guard condition is simply a first order logic formula, with only constant and free variables and without quantifiers. The only predicate symbols we use are the arithmetic comparison operators: $<, >, \leq, \geq, =, \neq$. In our case, we define two transition functions : T_c which is the transition function for transitions fired by completion event and T_{tick} which is the transition function for transitions fired by *tick* event.

Because we do not have the disjunction of the guards of all outgoing transitions of a state, T_c and T_{tick} could possibly return $\emptyset$ if no guard condition matches. Note that return $\emptyset$ or return to the same state as the state passed as a parameter are not the same. Return $\emptyset$ means no transition was taken, return to the same state means a reflexive transition was taken. In addition, because a state could have an entry action, to take a reflexive transition causes the execution of the entry action.

We also define $A : S \times V \to V$, the action function. A represents the execution of all entry actions. In fact, for each variable assignment and a state, A returns a new variable assignment.

With the definition of the two previous functions, modelling the run-to-completion step (RTC) in our cyclic state machine is quite easy. As we have two kinds of transition functions, we have two kinds of run-to-completion. We define $rtc_c : S \times V \to S \times V$ the run-to-completion step for completion event and $rtc_{tick} : S \times V \to S \times V$ the run-to-completion step for *tick* event. Each of them consists of: first apply T to the current state; second if T returns $\emptyset$ stay in the same state do nothing and return $\emptyset$, else apply A and return the new state and the new variable assignment.

The way to call these two kinds of run-to-completion is specific to the cyclic behaviour of our state machine. We define $Cycle : S \times V \to S \times V$ the function which represents the behaviour of a state machine in one cycle. At the beginning of *Cycle*, the state machine is in a stable state

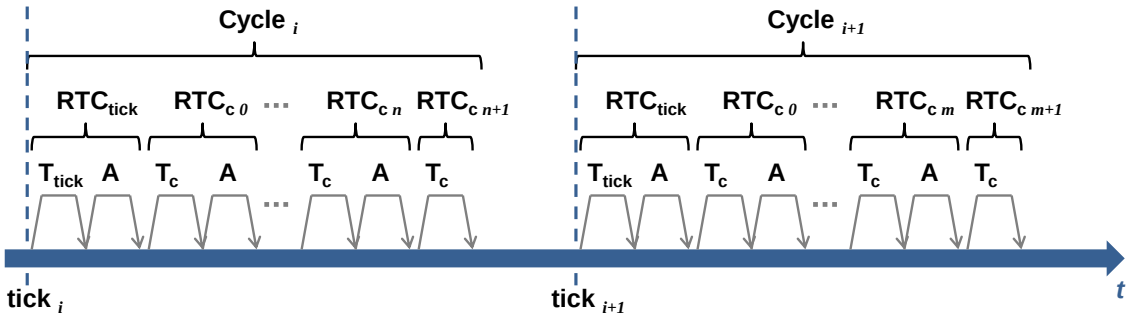

Figure 2: *Example of two consecutive cycles within a state machine*

configuration. *Cycle* first calls rtc_{tick} to deal with the *tick* event. If rtc_{tick} returns $\emptyset$, it means that no transition has been fired, so *Cycle* returns Id, the identity. If it does not return $\emptyset$, it calls rtc_c until there is no more transition with a completion event trigger to fire, i.e. rtc_c return $\emptyset$. In a more formalised way, we have :

$$Cycle = \begin{cases} rtc_{tick} \circ rtc_c^n & \text{with } n \in \mathbb{N} \text{ and } rtc_c^{n+1} \text{ returns } \emptyset \\ Id & \text{if } rtc_{tick} \text{ returns } \emptyset \end{cases}$$

When a cycle begins, the event pool is empty before one unique *tick* event occurs. If this *tick* event fired a transition, the *tick* event is consumed and the run-to-completion step will fill the event pool with one completion event. This completion event is processed by a new run-to-completion step. If a transition is fired, the event is consumed and the run-to-completion step will fill the event pool with a new completion event. The state machine will repeat the same mecanism until no more completion event are present in the pool (it corresponds to the iterative call of the RTC_c function in our semantics). Indeed, if no transition is fired, the completion event is discarded and the pool is left empty until the next cycle. Thanks to the synchrony hypothesis described in the previous section, we are sure that this chain of run-to-completion step will end before the next cycle i.e. before a new *tick* event occurs. Consequently, at the beginning of a cycle, the event pool is always empty before the *tick* event occurs and the completion event will only occurs after the processing of this *tick* event.

The initial state s_0 is a particular state in UML. It is a pseudostate. As defined in the UML semantics, this state has no trigger and guard defined on its unique outgoing transition. As it is particular in UML, its processing will be defined separetely of the other states in our semantics. We define a function $Cycle_0 : \{s_0\} \times \{v_0\} \to S \times V$. We have $Cycle_0 = rtc_c^n$ with $n \in \mathbb{N}^*$ where rtc_c^{n+1} returns $\emptyset$. This function is only called once at the very beginning of the execution of the state machine.

A cyclic state machine is defined by a $6-$tuple $\langle s_0, v_0, Cycle_0, S, V, Cycle \rangle$.

For example, Figure 2 describes two consecutive cycles in a state machine. Note that it is a particular case, since there is at least one rtc_c for each cycle.

3. Formal verification from model to code

3.1. Our method

In MDE, models are used all along the development chain and allow users to generate the source code implementation. Although a part of our work takes place at the code level, our contribution

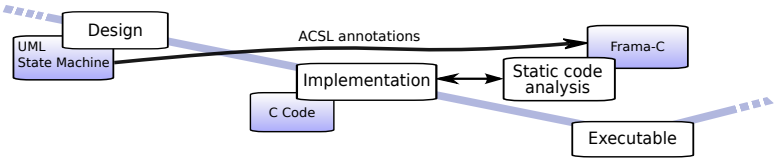

Figure 3: *Our process: verification from model to source code*

does not deal with code generation. We want to verify the behaviour of a C program, written by humans or machines, according to its model based design. We will only give the implementation pattern of a state machine, since information is needed on the code structure to manage our verification. Our method focuses on the use of the semantics of UML state machines to derive annotations to verify the code using static analysis.

Regarding the whole process in which we propose our verification method, we can compare our method with a code generation method. At a technical level, automation of properties generation for verification purpose is similar to automation of code generation. But, placed in a certification context like DO-178C for the aeronautical domain, the qualification constraints of a verification tool are much lighter than those of a code generator tool. If the verification tool fails, it does not introduce errors in the target software while a code generator might. A code generator must be qualified at least at the same level of criticality than the target software; it is not the case for a verification tool.

To conduct static analysis we use the Frama-C[6] framework. It is an open-source and modular environment which groups many different techniques and tools to conduct such analysis on C code. It is based on the ACSL language [3] (ANSI/ISO C Specification Language). ACSL is a specification language to express behavioural properties on C code. It is based on first order logic and allows to specify function contracts, invariants, variants, loop specifications, logic specifications and ghost codes. ACSL annotations are represented as comments in C code, using specific tags to be recognised by Frama-C. These annotations are without side effect on the program.

We use our cyclic UML state machine model to generate the corresponding ACSL function contracts to verify the code behaviour. An overview of our process is given Figure 3.

A function contract is composed of preconditions and postconditions. The function contract is: if the preconditions are true when the function is called then the postconditions must be true after the function execution. We use a Frama-C plugin named WP[7] to verify function contracts. WP is based on the weakest Precondition Calculus introduced in [7]. The weakest Precondition Calculus consists in computing the weakest precondition ensuring the postconditions. WP computes the weakest precondition of the function contracts and generates proof obligations for the verification of the implication of the weakest precondition by the initial preconditions. These proof obligations are discharged by solvers available through Frama-C.

In this work, we only focus on the verification of the transition functions implementation. They represent the core of a state machine behaviour.

3.2. State machine implementation pattern

To generate function contracts on the source code, we need to know: the prototype of functions; the name and the type of the variables of the program. In addition, function contracts are also

[6] `http://frama-c.com/`
[7] `http://frama-c.com/wp.html`

linked to the structure of the implementation. Therefore, we propose a code design pattern for the implementation of our state machines.

Although we only focus on the transition functions, we give a global code design pattern in order to give an overview of the implementation. This implementation pattern is based on a representation of the states as an enumeration type named `State`. The enumeration possible values of `State` are all the possible states of the state machine and one value named `Null`. This value will represent the $\emptyset$ used in our semantics. In addition, all the variables used in the model retain their names in the implementation.

The other parts of the implementation pattern of our state machine is composed of the following functions.

- Two transition functions, one for the tick event, namely `T_tick`, and one for the completion event, namely `T_c`. They represent the choice of the transition that will be fired according to the transition guards. It returns the targeted state if a transition has been fired, the `Null` value if not. Transition functions are, at top level, a `switch/case` structure to match with the current state. For each case, a conditional structure `if/else` is implemented for each outgoing transition of the state triggered by the corresponding event. It represents the guard of the outgoing transition. The code design pattern for the `T_tick` is given in listing 1 (`T_c` is based on the same pattern).

```
State T_tick (State current_state) {
  State output_state=Null;
  switch(current_state) {
    case state1 :
      if (condition_transition1)
        output_state=targeted_state;
      else if (condition_transition2)
        output_state=other_targeted_state;
    break;
  }
  return output_state;
}
```

Listing 1: *`T_tick` function pattern*

- An action function, namely `A`, which, for each state, executes the entry actions of the state. Note that, according to the semantics, `A` will only be executed if a transition has been fired (we do not give the code pattern of `A`).

- Two run-to-completion functions, namely `RTC_tick` and `RTC_c`, one for each possible event. Each one calls its corresponding transition function. The code pattern of `RTC_tick` is given in listing 2 (`RTC_c` is based on the same pattern).

```
State RTC_tick (State current_state) {
  State compute_state=T_tick(current_state);
  if (compute_state!=Null) {
    A(compute_state);
    return compute_state;
  } else return Null;
}
```

Listing 2: *`RTC_tick` function pattern*

- A function `Cycle` which implements the running of a state machine during one cycle. It calls first the run-to-completion function for the tick event. If the return is not the `Null` value: first it calls the run-to-completion function for the completion event until the return of `Null` and then it returns the new state computed. The termination of the function must be ensured

at the model level i.e. the model based design must guarantee that it exist a point where no further completion transition can be fired during the cycle. The code pattern of `Cycle` is given in listing 3. According to the semantics, we define, on the same pattern, a function `Cycle_0` which only calls the `RTC_c` function.

```
State Cycle (State current_state) {
  State compute_state=RTC_tick(current_state);
  if (compute_state!=Null) {
    State last_state;
    while(compute_state!=Null){
      last_state=compute_state;
      compute_state=RTC_c(last_state);
    }
    return last_state;
  } else return current_state;
}
```

Listing 3: *`Cycle` function pattern*

The running of the state machine is represented by a `while loop`. In each loop, the program waits until the next cycle and calls the cycle function (the code pattern is given in Listing 4).

```
current_state=Cycle_0(starting_state);
while(1) {
  wait_tick();
  current_state=Cycle(current_state);
}
```

Listing 4: *`while loop` pattern*

The application of the implementation pattern on the `T_tick` transition function of the example in Figure 1 is given in Listing 5.

```
State T_tick(State current_state){
  State output_state=Null;
  switch(current_state) {
    case DefaultPosition:
            if (pilot_lever==UP && squat_switch==AIR)
                output_state=WaitingForTakeoff;
    break;
    case WaitingForTakeoff:
            if (timer>=2 && squat_switch==AIR)
                output_state=StartRaisingGear;
            else if ((pilot_lever==DOWN && timer<2)||squat_switch==GND)
                output_state=DefaultPosition;
    break;
    case RaisingGear:
            if (pilot_lever==DOWN) output_state=LoweringGear;
            else if (pilot_lever==UP && up_switches==OK)
                output_state=GearUp;
    break;
    case GearUp:
            if (pilot_lever==DOWN) output_state=StartLoweringGear;
    break;
    case LoweringGear:
            if (pilot_lever==UP) output_state=RaisingGear;
            else if (pilot_lever==DOWN && down_switches==OK)
                output_state=GearDown;
    break;
  }
  return output_state;
}
```

Listing 5: *`T_tick` implementation for the LandingGear example*

3.3. Behavioural properties as function contract

The source code behavioural verification aims at proving properties stem from the UML state machine specification. At the code level, we define them as ACSL function contracts on the implementation. These function contracts aims at being generated from the state machine. The behavioural properties are divided in two categories. First, the specification completeness, "the specification is fully implemented"; second the specification soundness, "only the specification is implemented".

To ensure the specification completeness at the transition functions level, the implementation must ensure the following properties:

(a) for the current state of the state machine, if the transition guard is true, the transition function returns the specified targeted state;

(b) the transition function is without effect on state machine variables.

In the same vein as [9], we define an ACSL annotation pattern for each property. Property (a) is represented as one ACSL *ensures* clause for each possible outgoing transition of the current state. An *ensures* clause represents a property that must be true after the program execution. It corresponds to a postcondition. In fact, we generate a set of *ensures* clauses for each transition functions, `T_tick` and `T_c`. Each set regroups *ensures* clauses for each outgoing transition and for each state, according to the event handled by the transition function. Following the transition function prototype, the pattern for this property for a state is given in Listing 6. Note that the return of a function is defined by the keyword *result* in ACSL.

```
ensures <guard of outgoing transition 1>
   .   ==> \result == <target state of outgoing transition 1>;
   .
   .
ensures <guard of outgoing transition N>
     ==> \result == <target state of outgoing transition N>;
```

Listing 6: *Property (a) pattern*

Property (b) is represented by an ACSL *assigns* clause. The *assigns* clause is used to specify exhaustively the memory allocations possibly modified by the C program. So if it is specified with the keyword *nothing*, the clause guarantees that no memory allocation has been modified. Following the transition function prototype, the pattern for this property is given in Listing 7.

```
assigns \nothing;
```

Listing 7: *Property (b) pattern*

Specification soundness means that nothing else except the specified transitions is implemented in the program. To ensure soundness, the implementation must verify the following properties:

(c) for the target state resulting of the firing of a transition and its specified source state, the guard of the corresponding transition must be true;

(d) if no guard of the outgoing transitions of the current state is true, no transition is fired i.e. the transition function returns $\emptyset$.

Property (c) allows to verify that no unspecified transition exists between two states linked by a specified transition. As for property (a), it corresponds to one *ensures* clause for each possible outgoing transition of each state, according to the event handled by the transition function. The pattern for this property is given in Listing 8.

```
ensures \result == <target state of outgoing transition 1>
   .         ==> <guard of outgoing transition 1>;
   .
   .
ensures \result == <target state of outgoing transition N>
           ==> <guard of outgoing transition N>;
```

Listing 8: *Property (c) pattern*

Property (d) allows to verify that for a given state, there is no other possible target state than the specified ones. It is also represented as an *ensures* clause. The pattern for this property is given in Listing 9. The negation is expressed as the "!" symbol in ACSL.

```
ensures (!<guard of outgoing transition 1>
         && ...
         && !<guard of outgoing transition N>)
                    ==> \result == Null;
```

Listing 9: *Property (d) pattern*

All postconditions presented must be defined for each possible state. ACSL gives the possibility to define multiple named function contracts, called *behavior*, for a function. Therefore, we define, for the global function contract of each transition function, as many *behavior* as there are states with outgoing transitions triggered by the event handled by the transition function. In these *behavior*, the precondition deals with the current state. It is expressed as an *assumes* clause in ACSL. An *assumes* clause represents the property that must be true for applying the *behavior*. For instance, Listing 10 gives the ACSL *behavior* for the state RaisingGear of the `T_tick` function.

```
behavior RaisingGear:
   assumes current_state==RaisingGear;
   assigns \nothing;
   ensures (pilot_lever==DOWN) <==> \result==LoweringGear;
   ensures (pilot_lever==UP && up_switches==OK) <==> \result==GearUp;
   ensures (!(pilot_lever==DOWN) && !(pilot_lever==UP && up_switches==OK))
              ==> \result==Null;
```

Listing 10: *The behavior for the state RaisingGear in the `T_tick` function*

In addition, we need to add a property in the soundness category:

(e) if a state is not handled by the verified transition function (i.e. this state has no outgoing transition triggered by the event handled by the transition function), the transition function does not fire any transition.

Property (e) means that the return value of the transition function must be $\emptyset$ for all unhandled states. In ACSL, it is represented by a *behavior* composed of an *assumes* clause representing all the states not handled by the transition function and an *ensures* clause representing the `Null` value returned by the function. The example for the `T_tick` function of the example in Figure 1 is given in Listing 11.

```
behavior OtherStates:
   assumes current_state!=LoweringGear
           && current_state!=DefaultPosition
            && current_state!=WaitingForTakeoff
             && current_state!=RaisingGear
              && current_state!=GearUp;
   assigns \nothing;
   ensures \result==Null;
```

Listing 11: *property (e) for the `T_tick` function*

All the *behavior* described below represent the global function contract of a transition function. Each global function contract allows to check the conformity of each transition function with the behaviour expressed in the state machine. But, although we are able to detect unspecified transitions, we cannot detect dead code i.e. transitions that never happen at execution or states never reached.

Concerning the behavioural verification of the other implemented functions (*A*, *RTC*, *Cycle*), the verification follows the same principle. We want to verify them using ACSL annotations derived from the semantics of the state machine. But their verification raises other problems that the one presented here like for instance the verification of the loop described in *RTC* or *Cycle* function. We will not abord these subjects in this paper.

4. Our tool

We have implemented our approach in a prototype in Java. It comes as an Eclipse[8] plugin depending on the Topcased[9] framework. It allows, from the model explorer of a Papyrus[10] UML model, to choose a state machine and to generate ACSL contracts from it. Users only have to give the path to the C file they want to annotate in order to generate the annotated C file. We implement for each ACSL clause and ACSL structure we use, a corresponding object for which we implement its string representation. For instance, in our case an ACSL *behavior* is composed of an object AssumesClause, an object AssignsClause and a collection of objects EnsuresClause.

The generation is done in 4 steps. First, we check that the selected state machine is well formed according to our model rules. Indeed, the Topcased Papyrus editor allows the creation of UML models based on the whole standard, but we only use a subset in our case. Secondly, we check that the C file contains the transition functions. We use the Eclipse CDT API[11] to parse the C file and to retrieve the corresponding function to annotate. We memorise their locations in the file thanks to their offsets. Thirdly, we parse the state machine and we create all the behaviours for each state. Finally we generate a C file, corresponding to the C code and the annotations generated at the right places in the code.

For the example described in Figure 1, we are able to generate 55 lines of function contracts for more than 40 lines of C code for the two transition functions. All the function contracts have been verified in a few seconds thanks to Frama-C and its plugin WP.

5. Related work

Some work exists on the verification of source code using annotations generated from a model specification.

[8] proposes a way to automatically annotate C code according to a specification composed of SAM (Structured Automata Model) automata. SAM is a domain specific language for the behavioural representation of avionic components. The authors present an algorithm to generate annotations from SAM automata to verify the code behaviour. The approach is similar to our own since it consists in generating function contracts on the transition function implementing the SAM automaton and they also present an industrial experimentation with promising results. By

[8] `www.eclipse.org`

[9] Toolkit in OPen-source for Critical Application and SystEms Development. It offers Model Driven Engineering activities and it is based on the Eclipse environment. `www.topcased.org`

[10] It is tool for modelling in UML. The current Topcased model editor are based on Papyrus version 0.8. `www.papyrusuml.org`

[11] C/C++ Development Tooling. `www.eclipse.org/cdt/`

contrast, the SAM automaton and generated function contracts are less complex than our UML state machine and our annotations.

The Aorai plugin of Frama-C [20] allows to generate ACSL annotations from an automaton specification expressed in LTL (Linear Temporal Logic). Aorai automatically annotates the targeted source code and the verification is performed using the solvers available from Frama-C. Actually, the automaton specification represents a chain of function calls and function returns. Each of them can be associated to properties on the program variables. At the end, if the annotations are verified, then the source code conforms to the specification. Aorai focuses on function calls at global program level while our method focuses on function behaviour. Moreover, the specification in LTL is more complex and less intuitive than a specification modelled with state machine diagrams for users non-familiar with temporal logic.

In [16], authors propose the theoretical foundations of a toolset to generate annotations on the software implementation from control theory properties and proof expressed in a control systems design. The goal is to obtain an autocoder with proofs. The source language is an open-source alternative of Matlab[12], Scilab[13] and the target code is implemented in C language. The properties annotated in the design are translated in ACSL annotations in the code. The ACSL annotations are then verified using Frama-C. The authors present two methods. One is a direct mapping of the annotations on the design and the semantics of Scilab operators to annotations on the code. The other uses a gateway language, Lustre [14] in order to take into account different front end languages for the design. The spirit of the approach is close to ours. It differs by the type of systems to verify, the design language and the properties to verify.

In our approach, we use a part of the UML language version 2.4.1 limited to state machine modelling as source for the design. As the current UML semantics is semi-formal, we needed to formalise its semantics in order to avoid any ambiguities and to use formal methods in a rigorous way. In the particular case of UML state machines, there is a lot of work on the formalisation of their semantics. [6] aims at giving an overview of the state of the art. It lists 26 semantic approaches structured in three categories. First, it lists work which is based on standard mathematical concepts and notations. For instance, [18] uses Labelled Transition Systems (LTS) expressed in an algebraic specification language for the representation of the semantics and [5] uses Abstract State Machines (ASM). Secondly, it lists the approaches expressing the semantics as a set of rewriting rules. For example, [21] and [12] use graph transformations and [2] defines translation rules to map an UML specification to high-level Petri nets. Finally, it groups approaches based on the translation of UML state machines into other formal languages. For instance, [1] defines the semantics in PVS (Prototype Verification System) and [17] presents a global semantics which is implemented in PROMELA for model checking. Note that none of the approaches supports all the UML state machines concepts. Our work is clearly in the first category. We define a very simple mathematical semantics dedicated to the needed concepts. Our semantics uses new concepts (like cycle, transition functions, etc.), but it is fully compliant with the semantics described in the UML standard.

6. Conclusion

We presented a method to automatically verify the behaviour of a C source code with respect to its UML design model. The main advantage is to have a full MDE process which gives access to formal methods and associated tools to non-expert users. The main drawback of the approach we presented is that the implementation is very close to the semantics of our state machines.

[12] www.mathworks.fr/products/matlab/

[13] www.scilab.org

This work was motivated for multiple reasons. It allows users to be more confident on their implementation in a simpler way: as the annotations are automatically generated from the model and automatically verified, users do not need to change their technical know-how. Furthermore, although MDE already permits to generate tests on the code, static analysis is more exhaustive than software testing. Indeed, static analysis does not just test the code, it proves it i.e the results of the verification are valid for all possible executions.

The results of our method are promising but it needs to be improved and experimented on more complex models. Here, we only test it on small exemples (dozen of states). Moreover, we are thinking about its application on other implementation patterns. In further work, the feedback of the verification must be adapted to help the user to correct the implementation errors. Currently, the user relies on the verification results of each annotation to determine where the problem is on the code. We could explicit these feedback in a more detail and user-friendly way or present them at model level. Furthermore, we plan to extend the UML subset we used. Our state machines are limited to simple states. We would like to take into account hierarchical states such as composite states or submachine states, as defined in the UML standard. We also need to define the formal semantics of the subset in a more complete way since we only presented here the key concepts useful for our method. Therefore, we limited our contribution for this paper to the behavioural verification of transition functions. In a very close future, we will work on the way to verify the other functions defined in our semantics, as this point is mandatory to obtain a complete proof of the compliance of the implementation with the state machine behaviour. Finally, we plan to make the annotation generator we presented available online to get the feedback of the community.

References

[1] Demissie B. Aredo. Semantics of uml statecharts in pvs. In *In Proc. of the 12th Nordic Workshop on Programming Theory (NWPT'00)*, 2001.

[2] L. Baresi and M. Pezzè. On formalizing uml with high-level petri nets. In GulA. Agha, Fiorella Cindio, and Grzegorz Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 276–304. Springer Berlin Heidelberg, 2001.

[3] P. Baudin, P. Cuoq, J.C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL Version 1.6*, 2012.

[4] GÃl'rard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[5] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the dynamics of uml state machines. In Yuri Gurevich, PhilippW. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer Berlin Heidelberg, 2000.

[6] Michelle L. Crane and Juergen Dingel. On the semantics of uml state machines: Categorization and comparision. In *In Technical Report 2005-501, School of Computing, Queen's*, 2005.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[8] S. Duprat, P. Gaufillet, V. Moya Lamiel, and F. Passarello. Formal verification of sam state machine implementation. In *ERTS*, France, 2010.

[9] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420, 1999.

[10] A. Fernandes Pires, S. Duprat, T. Faure, C. Besseyre, J. Beringuier, and J-F. Rolland. Use of modelling methods and tools in an industrial embedded system project : works and feedback. In *ERTS*, France, 2012.

[11] R. France, A. Evans, K. Lano, and B. Rumpe. The uml as a formal modeling notation. *Comput. Stand. Interfaces*, 19(7):325–334, November 1998.

[12] Martin Gogolla and Francesco Parisi Presicce. State diagrams in uml: A formal semantics using graph transformations - or diagrams are nice, but graphs are worth their price. In *University of Munich*, pages 55–72, 1998.

[13] Martin Gomez. Embedded state machine implementation. *Embedded Systems Programming*, page 41, 2000.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[16] Romain Jobredeaux, T.E. Wang, and E.M. Feron. Autocoding control software with proofs i: Annotation translation. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7C1–1–7C1–13, Oct.

[17] Johan Lilius and Iván Porres Paltor. Formalising uml state machines for model checking. In Robert France and Bernhard Rumpe, editors, *ÂńUMLÂż'99 âĂŤ The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–444. Springer Berlin Heidelberg, 1999.

[18] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing uml active classes and associated state machines - a lightweight formal approach. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin Heidelberg, 2000.

[19] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and DennisR. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer Berlin Heidelberg, 2009.

[20] Nicolas Stouls and Virgile Prevosto. *Aoraï Plug-in Tutorial*.

[21] Dániel Varró. A formal semantics of uml statecharts by model transition systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer Berlin Heidelberg, 2002.