Use of formal methods in embedded software development: stakes, constraints and proposal

Anthony Fernandes Pires*†, Thomas Polacsek†, Virginie Wiels† and Stéphane Duprat*
*Atos Intégration SAS, 6 impasse Alice Guy, B.P. 43045, 31024 Toulouse cedex 03, France
†ONERA, 2 avenue Edouard Belin, 31055 Toulouse, France

***Abstract*—In aeronautics, software development is submitted to strong constraints. The DO-178 certification standard specifies development and verification objectives. Moreover, its supplement DO-333 defines guidelines for the use of formal methods in this context. Formal methods are used in industry for different purposes and often require the intervention of experts for their processing. In this paper, we propose an approach to answer a certification objective using formal methods while keeping them usable for non-experts. We present an automatic method to check the compliance of a C source code according to its Low Level Requirements expressed as an UML state machine and we show how it addresses objectives of the DO-333.**

Keywords: Certification, aeronautics, formal methods, model driven engineering, verification and validation

I. Introduction

DO-178/ED-12, *Software Considerations in Airborne Systems and Equipment Certification* [9], is the current basis for software assurance in the civil aeronautical domain. Version B of this standard was published in 1992. It has been updated recently into version C [10] and includes technical supplements to take into account and facilitate the appropriate use of new software engineering techniques that have emerged since 1992. DO-333/ED-216 [11] is the formal methods supplement. Formal methods can be applied to many of the development and verification activities required for software. The supplement proposes guidance for the use of formal methods. It describes the activities that are needed when using formal methods, new or modified objectives and evidence needed for meeting those objectives.

In this paper, we will present a formal verification approach for avionics software and position it with respect to DO-333. Section 2 briefly presents DO-178 and DO-333. Section 3 synthesizes our observations on the use of formal methods in industry. Section 4 deals with our proposal. Section 5 presents our tool and Section 6 provides the links between our approach and the DO-333 objectives. Section 7 concludes the paper.

II. DO-178 and DO-333

A. DO-178

DO-178 does not prescribe a specific development process, but instead identifies important activities and design considerations throughout a development process and defines objectives for each of these. DO-178 distinguishes development processes from integral processes that are meant to ensure correctness, control, and confidence of the software life cycle processes and their outputs. The verification process is part of the integral processes along with configuration management and quality assurance.

Four processes are identified as comprising the software development processes in DO-178:

1) The software requirements process develops High Level Requirements (HLR) from the outputs of the system process;
2) The software design process develops Low Level Requirements (LLR) and Software Architecture from the HLR;
3) The software coding process develops source code from the software architecture and the LLR;
4) The software integration process loads executable object code into the target hardware for hardware/-software integration.

Each of these processes is a step towards the actual software product.

The results of the four development processes must be verified. Detailed objectives are defined for each step of the development, with some objectives defined on the output of a development process itself and some on the compliance of this output to the input of the process that produced it. Figure 1 presents the verification objectives and activities in relationship with the development artifacts. For example, LLR shall be accurate and consistent, compatible with the target computer, verifiable, conformed to requirements standards, and they shall ensure algorithm accuracy. Furthermore, LLR shall be compliant and traceable to HLR.

DO-178 identifies reviews, analyses and test as means of meeting these verification objectives. Reviews provide a qualitative assessment of correctness. Analyses provide repeatable assessment of correctness. Reviews and analyses are used for all the verification objectives regarding HLR, LLR, software architecture and source code. Test is used to verify that the executable object is compliant with LLR and HLR. Test is always based on the requirements (functional test) and includes normal range and robustness cases.

B. DO-333

A formal method is defined as a formal analysis carried out on a formal model. This perspective is important because it permits discussion of formal methods according to the major life cycle processes called out in DO-178, especially development and verification processes. Development processes are applicable to formal models, and verification processes are applicable to formal analyses.

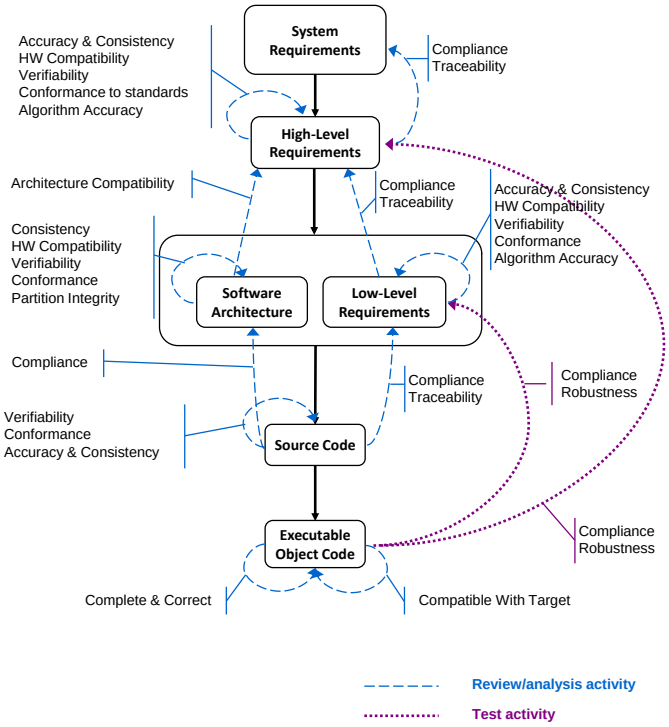

Fig. 1. Verification objectives and activities

With formal analysis, the correctness of life cycle data with respect to a formal model or property can generally be proved or disproved; therefore, formal analysis is able to replace the conventional methods of review, analysis, and test, as specified in DO-178, for some verification objectives. DO-333 guidance details the potential use of formal methods at each development level and gives the conditions for the use of a given formal analysis for a given verification objective.

III. Formal Methods in Industry

Without any methodological constraints like DO-178, adopting formal methods is motivated by industrial expectations that are cost reduction and quality improvement. Formal methods are very often a way of obtaining a high degree of quality earlier and as a consequence, a way of reducing efforts of verification in later stages of tests and maintenance.

Formal methods are already used in industry. For instance, [12] presents the industrial experience of formal verification techniques for avionics software products at Airbus. In their context, these techniques are used for proving properties on the code, for analyzing Worst Case Execution Time and for computing the maximum stack usage. Moreover, foreseeable use of formal verification techniques for other tasks is also presented, like the proof of absence of runtime errors. This use is reported in more recent work [1]. Another example of the use of formal methods is described in [8]. The authors present a case study on the formal verification of industrial code at Dassault Aviation. Furthermore, they report the link of the application of these techniques with certification objectives in [1]. More examples on the use of formal methods in industry can be found in [1], [2] and [7].

Whatever the motivations and the context, we observed in different projects at Atos that introducing formal methods in the process often necessitates the participation of experts. First, we observed that experts can participate directly in the project in order to perform activities with high added-value. Secondly, we observed that experts can be in charge of the training of the team that will use formal techniques and subsequently provide some support to the team. A mix of these two use cases is not forbidden as a first iteration in a starting project can be done by experts and continued by the development team.

IV. Our Proposal

A. Context

Our goal is to integrate formal verification in an existing process, in the easiest way for the development teams. We want both to answer the certification objectives of DO-178 by using formal methods and to remain in the technical knowledge of development engineers which are not necessarily expert in formal methods.

We focus on the verification objective of the compliance of the executable object code to LLR. To achieve this goal, DO-333 proposes alternative verification paths as shown on Figure 2 in replacement of tests: compliance of executable object code to LLR can be demonstrated using formal analysis on source code and analysis of property preservation between source code and executable code.

In this paper we only target the compliance of source code with respect to LLR using formal analysis. The formal analysis of the source code can be managed by static analysis. Indeed, static analysis aims at analyzing a program without executing it. In particular, it is possible to use static analysis to verify annotations on the code using deductive proof. These annotations are assertions which can express behavioural properties, variant, invariant, etc. Deductive proof is a formal technique allowing proving assertions on the code by deduction.

B. Principle

Our approach is simple: based on formal techniques, we are looking to automatically derive from the LLR, annotations to be proved on the code. These annotations are then automatically verified by formal verification using deductive proof. Due to our industrial context, we choose specific technologies: to express the LLR, to implement the source code and to conduct the formal verification.

For expressing the LLR, we choose the UML standard. UML is a modelling language, widespread and well accepted in development teams. Here, we focus on its use for the expression of the software behaviour through state machines. For the source code language, we choose the C language and we introduce specific code patterns. For the formal verification, we choose the Frama-C framework[1]. It is a free and open-source tool. In Frama-C, the annotations on the code are expressed in ACSL. ACSL (ANSI/ISO C Specification Language) is a behavioural specification Language for C code. It is based

[1] `frama-c.com`

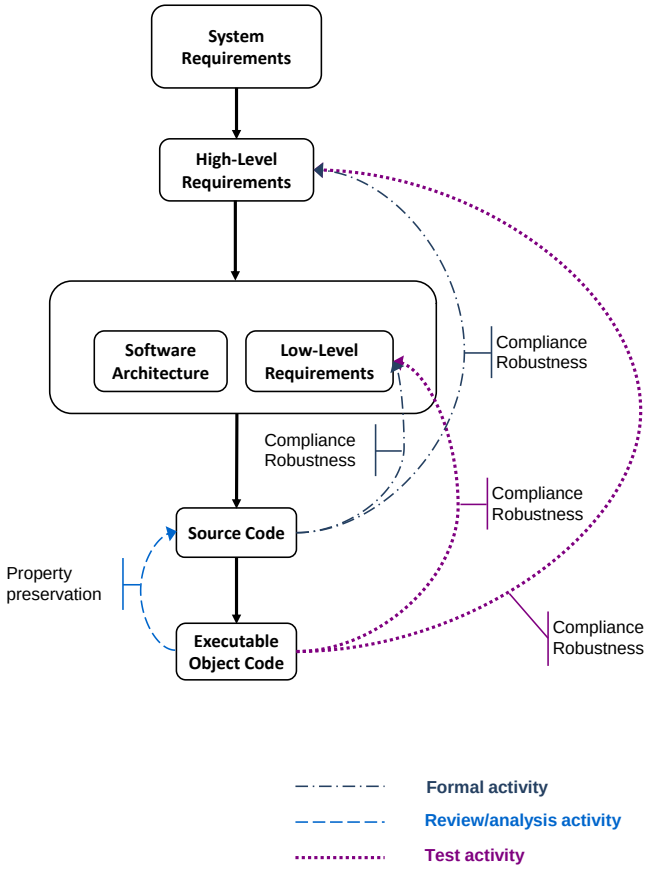

Fig. 2. Alternative verification paths

on first order logic and it allows defining properties on the implementation without side effects.

A summary of our proposal is available Figure 3. We detailed the different stages in the next paragraphs.

C. The expression of LLR

We use a subset of UML adapted to our needs and restricted to a part of the state machine model to express LLR. We consider state machines run by a clock and composed of simple states. At each clock tick, the state machine does a number of actions and then waits for the next clock tick. This stage is called a cycle. Actions are synchronous and only authorised in the entry behaviour of a state.

In our state machines, transitions can have a trigger, linked to an event, and a guard to control its firing. furthermore, transitions have no effects. The trigger of a transition can be linked to two events, the tick event which represents a clock tick, or the completion event which is the default event in UML. A completion event is automatically generated at the end of the actions defined in a state or at its entry if no action is defined. In our subset, this event must be used as trigger for transitions linking two states occurring

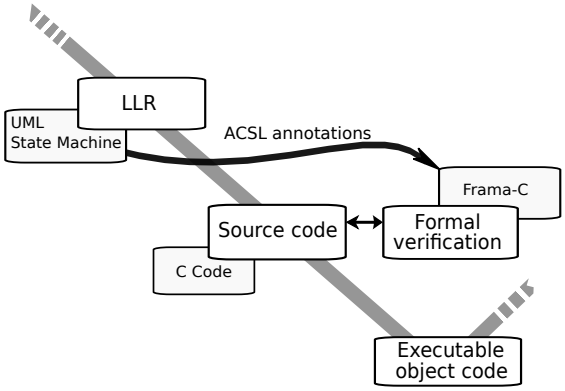

Fig. 3. Our proposal

during the same cycle. The guard of a transition is a simple boolean expression. This expression is composed of variables, constants, boolean and arithmetic operators but no quantifier. Moreover, no function can be used in a guard. Finally, we authorise only one pseudostate: the initial state.

We add two constraints to our subset. The first one is that every processing starting within a clock tick ends before the next clock tick. The second one is that our state machines are deterministic: we do not authorise conflicting transitions.

In addition, we define a formal semantics compliant with the semi-formal semantics defined in the UML standard. We abstract the notion of event by using specific functions to define the processing of each event. Here, we give some clues about the formal semantics of the transition functions. For more details about the whole formal semantics, the reader could refer to [4].

We define T_c et T_{tick}, the transition functions which define respectively the processing of the completion event and the processing of the tick event. According to the source state, the event triggering the transition and the guard of the transition, these functions return a new state. Because we do not have the disjunction of the guards of all outgoing transitions of a state, these transition functions can return the value $\emptyset$, representing that no transition has been fired. We can note that to return $\emptyset$ or to return the same state as the source state is not the same. To return the same state means a reflexive transition was taken and the action defined in the state was executed.

D. The source code

In order to generate annotations on the code, we need information on it. We need to know how the program is structured, the prototype of each function and the name and the type of all the variables. So we propose a code pattern to follow for the implementation of our state machines.

Firstly, we impose that the names of states and variables defined in the model hold in the code. Secondly, we define the states as a specific enumeration type named `State`, and we define a specific variable to represent the current state named `current_state`. The value $\emptyset$ of our semantics is represented by the value `Null` in the code, defined within the enumeration type `State`. Finally, we map each function defined in the semantics by a C function in the code. The code patterns of all the functions are defined in [4].

E. The generated annotations

In formal verification, the annotations on the code are assertions representing the behavioural properties that this code must verify. These annotations can be expressed by function contracts on the code. A function contract is composed of preconditions and postconditions. Its meaning is: the postconditions must be true after the program execution, provided that the preconditions are true at its beginning.

Our approach allows generating two kinds of properties as ACSL function contracts from the LLR: the LLR completeness and the LLR soundness. LLR completeness ensures that the LLR are fully implemented. LLR soundness ensures that only the LLR are implemented.

In this paper, we focus on the generation of these properties for the transition functions. For each transition functions, the LLR completeness can be expressed by two behavioural properties:

(a) For the current state, if the guard of an outgoing transition is true then the transition function must return the state targeted by the transition ;

(b) The transition function does not modify the state machine variables.

The LLR soundness for a transition function can be expressed by three behavioural properties:

(c) For the current state, if a transition has been fired, i.e. the new state is returned by the transition function, then the guard of the transition must be true ;

(d) For the current state, if no guard of its outgoing transitions is true then the transition function must return nothing (represented by the `Null` value in our code pattern).

(e) If a state is not handled by the transition function then the transition function must return nothing, i.e. no transition is fired.

F. The formal verification

The generated function contracts are automatically transformed into proof obligations by the plugin WP[2] of the Frama-C framework and verified by the available solvers, like alt-ergo[3]. Then, the user can access the verification result for each annotation.

G. Example

We illustrate our approach with an example of the landing gear of an UAV (Unmanned Aircraft Vehicle). We express the LLR of the software with a state machine given in Figure 4.

The corresponding C code for this example is expressed using our code patterns. Their application for the implementation of the T_{tick} function of the example is given in Listing 1.

The completeness properties and the soundness properties for the transition functions are generated from the model

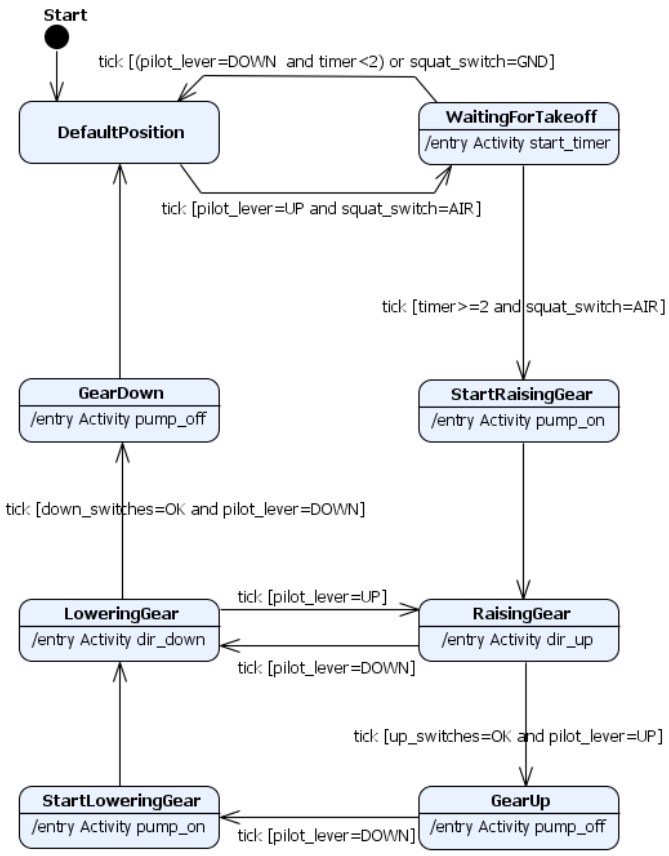

Fig. 4. Landing gear example

given in Figure 4 and expressed as ACSL function contracts on the code, one for each transition function. In ACSL, a precondition for a function contract is defined by a *requires* clause and a postcondition is defined by an *ensures* clause. Furthermore, a function contract can be decomposed as a set of ACSL *behavior*. An ACSL *behavior* represents one behaviour that the function must guarantee according to a condition. This condition is expressed as an *assumes* clause in ACSL. An ACSL *behavior* is also composed of preconditions and postconditions.

In our function contracts, the properties (a), (b), (c) and (d) are defined in an ACSL *behavior* on the code, one for each state handled by the transition function (a state is handled by transition function, if at least one of its outgoing transitions is triggered by the event associated with the transition function). The *assumes* clause of the ACSL *behavior* is about the value of the current state. The ACSL *behavior* is then composed of different *ensures* clause:

- The property (a) and (c) are merged in one *ensures* clause for each possible outgoing transition of the state. This *ensures* clause expresses that the transition function returns the targeted state of a transition if and only if the guard of this transition is true.
- The property (b) is defined by an ACSL *assigns* clause. This clause allows defining the memory allocations possibly modified by the function. It is defined using the ACSL keyword *nothing* meaning that no

[2] `frama-c.com/wp.html`
[3] `http://alt-ergo.lri.fr/`

```
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 1. T_{tick} implementation for the LandingGear example

memory allocation has been modified by the function.

- the property (d) is defined by an *ensures* clause representing that if no guard is true then the transition function returns `Null`.

The property (e) is defined by an ACSL *behavior* for the transition function. The *assumes* clause is about the value of the current state. The ACSL *behavior* is then composed of one *ensures* clause defining that the return of the transition function must be `Null` for this behaviour.

The full function contract for the T_{tick} function is given Listing 2. if we look at the first *behavior*, it concerns the state `DefaultPosition` of the example. The clause "`assigns \nothing;`" represents the property (b). As there is only one outgoing transition from this state triggered by the tick event, the property (a) and (c) are defined by the clause "`pilot_lever == UP && squat_switch == AIR) <==> \result == WaitingForTakeoff;`" (the *\result* keyword represents the return of the function in ACSL). The clause "`! (pilot_lever == UP && squat_switch == AIR) ==> \result == Null;`" represents the property (d). At last, the property (e) is represented by the *behavior* called `OtherStates`.

Regarding the verification of these function contracts, they have been verified in a few seconds using the Frama-C framework.

V. A TOOL SOLUTION, AGRUM

As we can generate annotations from the model, we can automate this generation in order to conceal the major part of the approach and bring this method to non experts. AGrUM (ACSL Generator from UML Model) is an Eclipse plug-in to automatically generate behavioural annotations from a UML state machine based design to C code. It is free and open-source. Technically, our plug-in takes advantage of Eclipse-based tool as Papyrus[4] for the design modelling of the software and Eclipse CDT[5] for the management of the C source code file. It is a prototype and it only manages the generation of the annotations for the transition functions, as described in this paper.

The use of the plug-in is quite easy. The user designs its software behaviour as state machines using the Papyrus model editor. He then just has to select its state machine in the model explorer, open the generation pop-up, select the C file to annotate and if its code pattern conforms to the state machine and the model respects the subset, the plug-in automatically annotates the C file. Then, the annotated C file can be processed by Frama-C and the user can obtain the result of the verification.

The AGrUM project is stored in EclipseLabs and it is available at: `http://code.google.com/a/eclipselabs.org/p/agrum/` Users can find the source code and an update site to automatically install the plug-in in an Eclipse SDK 3.7. Furthermore, the website provides information, the landing gear example presented in this paper and videos on its use. Note that this plug-in is also fully compatible with TOPCASED 5.2 or later.

VI. CERTIFICATION OBJECTIVES

Verification objectives and activities are given on Figure 1. They are attached to the four development processes described in section 2. In DO-333, objectives are synthesized inside tables given in annex of the document, these tables reference specific sections of the document where the objectives are defined. The four tables that are of interest for our approach are:

- Table FM.A-4: Verification of Outputs of the Software Design Process;
- Table FM.A-5: Verification of Outputs of Software Coding and Integration Processes;

[4] `www.eclipse.org/papyrus/`

[5] `www.eclipse.org/cdt/`

```
/*@behavior DefaultPosition :
    assumes current_state == DefaultPosition;
    assigns \nothing;
    ensures (pilot_lever == UP && squat_switch == AIR) <==> \result == WaitingForTakeoff;
    ensures  ! (pilot_lever == UP && squat_switch == AIR) ==> \result == Null;

behavior WaitingForTakeoff :
    assumes current_state == WaitingForTakeoff;
    assigns \nothing;
    ensures ( ( pilot_lever == DOWN && timer < 2 )  || squat_switch == GND) <==> \result == DefaultPosition;
    ensures (timer >= 2 && squat_switch == AIR) <==> \result == StartRaisingGear;
    ensures  ! ( ( pilot_lever == DOWN && timer < 2 )  || squat_switch == GND) &&  ! (timer >= 2 && squat_switch == AIR)
               ==> \result == Null;

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

behavior GearUp :
    assumes current_state == GearUp;
    assigns \nothing;
    ensures (pilot_lever == DOWN) <==> \result == StartLoweringGear;
    ensures  ! (pilot_lever == DOWN) ==> \result == Null;

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

behavior OtherStates :
    assumes current_state != DefaultPosition && current_state != WaitingForTakeoff && current_state != RaisingGear
                        && current_state != GearUp && current_state != LoweringGear;
    assigns \nothing;
    ensures \result == Null;
*/
State T_tick(State current_state){
            ⋮
```

Listing 2. Generated annotations from the model for the T_{tick} function

- Table FM.A-6: Testing of Outputs of Integration Process;
- Table FM.A-7: Verification of Verification Process results.

There are 10 tables in total, tables FM.A-1, FM.A-8, FM.A-9, FM.A-10 deal with planning, configuration management, quality assurance and certification liaison. Table FM.A-2 defines objectives for the software development processes and table FM.A-3 is about the verification of outputs of software requirements process.

In this section, we present the objectives addressed by our approach for the four targeted tables.

A. *Fulfilled objectives*

As mentioned previously, our approach targets the compliance of source code with respect to LLR. The goal is to replace test of the executable object code with respect to LLR. The main certification objective is thus the compliance of EOC (Executable Object code) with respect to LLR which is reached by taking the alternative path proposed by DO-333 via the source code. Other objectives are also addressed by our approach. We give the list below, they are very similar to the ones given in appendix B of [11]. Figure 5 synthesizes these objectives.

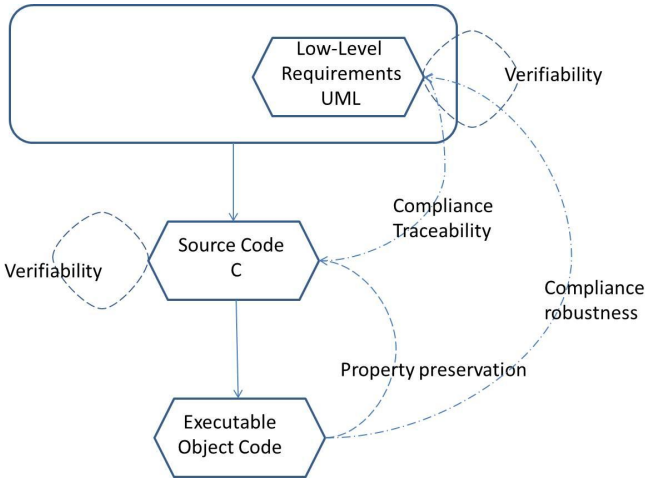

Fig. 5. Objectives addressed by our approach

For each objective, we give the reference of the objective inside the table, then we list the sub-objectives given in the corresponding sections and the associated justification for our approach.

a) Table FM.A-5, objective FM13: formal method is correctly defined, justified and appropriate. This supplemen-

tary objective must be addressed every time a formal method is used to achieve a verification objective. It is decomposed into three part detailed below:

- FM.6.2.1.a Formal notations: The syntax and semantics of the subset of the C programming language are mathematically defined and verified to be precise and unambiguous. We have also defined semantics for the subset of UML that is considered in our approach.
- FM.6.2.1.b Soundness: The verification technique used in Frama-C is based on Hoare logic that has been proved sound provided that the logic used for properties is sound [5].
- FM.6.2.1.c Assumptions: No specific assumption is used for the verification of the properties on the source code.

b) Table FM.A-4, Objective FM4: Low-level requirements are verifiable:

- FM.6.3.2.d Verifiability: The LLR are expressed in a restricted subset of UML state machines that can be translated into ACSL annotations and proved on the source code. Consequently, the LLR are verifiable.

c) Table FM.A-5, Objective FM1: Source code complies with low-level requirements:

- FM.6.3.4.a Compliance with LLR: The compliance of the source code with the LLR is achieved through the use of two tools: AGrUM which translates the UML LLR into ACSL annotations, and Frama-C which verifies that the source code satisfies the ACSL annotations. These two tools would have to be qualified with respect to DO-330 (Tool qualification document).
- FM.6.3.4.c Verifiability: Specific coding patterns are defined in our approach in order to ensure that the code will be compatible with the verification tool.
- FM.6.3.4.e Traceability: The verification that the source code satisfies all the LLR ensures that the LLR were developed into source code.

d) Table FM.A-6, Objective FM3: Executable object code complies with low-level requirements:

- FM.6.7.d Compliance with LLR: Formal analysis of source code is performed to reach this objective. Complementary analyses are necessary to show property preservation between source code and object code.

e) Table FM.A-6, Objective FM4: Executable object code is robust with low-level requirements:

- FM.6.7.b Robustness with LLR: Formal analysis of source code is performed to reach this objective. Complementary analyses are necessary to show property preservation between source code and object code.

B. Remaining objectives

The following objectives should also be addressed and are not yet addressed in our approach:

- Table FM.A-5, Objective FM10: Formal analysis cases and procedures are correct ;
- Table FM.A-5, Objective FM11: Formal analysis results are correct and discrepancies explained ;
- Table FM.A-7, Objective FM4: Coverage of low-level requirements is achieved ;
- Table FM.A-7, Objective FM5-8: Verification of software structure is achieved ;
- Table FM.A-7, Objective FM9: Verification of property preservation between source and object code.

The first three ones are not yet addressed but could easily be achieved by specific reviews. The two last ones are the ones that are more difficult to achieve, they necessitates specific analyses that remain to be defined. Objectives FM5 to FM8 in table FM.A-7 are the objectives that replace structural coverage objectives used when test is the verification method. For formal methods, the coverage objective is lifted to the requirement level, we have to demonstrate that the requirements that have been formally proven are complete with respect to the considered code. Objective FM9 of table FM.A-7 is added because formal verification is done on the source code and not on the executable object code, it thus remains to be shown that the properties verified on source code are still satisfied on executable object code. This objective can be achieved by reviews or using formal techniques [6].

VII. Conclusion

The approach we described is a proposal for verifying the compliance of a source code according to LLR, as an objective of DO-178. It does not aim at being a magical tool which resolves all the constraints of the use of formal methods but it aims at opening discussion and providing a way to plug formal methods in existing development processes, taking into account certification objectives and the technical knowledge of the development teams. Moreover, the main advantage of the approach is to have an automatic verification process with optimum quality.

Today, the approach covers only a part of the code, the verification of transition functions of the state machine. However, the behavioural specification is mainly defined in the transition functions, and partially in the function managing the call of the actions, for which the automatic verification is currently experimented. The other functions are based on the call of these leaf functions and their implementation is constant for all state machines. The proof of these functions is a bit more complex than the proof of transition functions because of the use of loops in their implementation. But once their proof realised, it will be available for every case. The automatic verification of these functions will be studied in a very close future.

The use of UML allows targeting a lot of users but our subset needs to be fully formalised to use the approach. Moreover, it would be meaningful to extend our subset to use the approach in efficient industrial cases. Indeed, It will be interesting to extend it to multi-events pattern and to the use of hierarchical states. We have seen in prior projects that hierarchical states, like submachine states, can be widely

used in the modelling of software [3]. We can note that we only use UML state machines in our approach. Current work at Atos aims at studying the use of UML activity for a similar approach. More precisely, this work focuses on the generation of ACSL annotations from UML Activity for automatic formal verification. A demonstration tool is already available at: `http://code.google.com/a/eclipselabs.org/p/a2acsl-project/`.

To finish, the verification results are still given as produced by the Frama-C tool but the retrieval of these results for a non expert, in a user-friendly way, is the subject of current work and should be integrated in the AGrUM tool. In the future, we would like to take advantage of the correspondance of code patterns with the modelling to display the possible errors detected by the deductive proof on the UML model.

REFERENCES

[1] Jean-Louis Boulanger. *Utilisations industrielles des techniques formelles: interprétation abstraite*. Hermès science publications-Lavoisier, 2011.

[2] Jean-Louis Boulanger. *Industrial Use of Formal Methods: Formal Verification*. John Wiley & Sons, 2012.

[3] Anthony Fernandes Pires, Stéphane Duprat, Tristan Faure, Cédrik Besseyre, Jack Beringuier, and Jean-François Rolland. Use of modelling methods and tools in an industrial embedded system project : works and feedback. In *Embedded Real-time Software and Systems (ERTS^2)*, France, 2012.

[4] Anthony Fernandes Pires, Thomas Polacsek, Virginie Wiels, and Stéphane Duprat. Behavioural verification in embedded software, from model to source code. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 320–335. Springer Berlin Heidelberg, 2013.

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

[6] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[7] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *Software, IEEE*, 30(3):50–57, 2013.

[8] Dillon Pariente and Emmanuel Ledinot. Formal verification of industrial c code using frama-c: a case study. In *Formal Verification of Object-Oriented Software*, 2010.

[9] RTCA/EUROCAE. DO-178B/ED-12B: Software Considerations in Airborne Systems and Equipment Certification, December 1992.

[10] RTCA/EUROCAE. DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification, 2011.

[11] RTCA/EUROCAE. DO-333/ED-216: Formal Methods Supplement to DO-178C and DO-278A, 2011.

[12] 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.