Formal Software Verification at Model and at Source Code Levels

Anthony Fernandes Pires, Thomas Polacsek, Stéphane Duprat

Formal Software Verification at Model and at Source Code Levels

Anthony Fernandes Pires[*,**], Thomas Polacsek[*], Stéphane Duprat[**]
[*] ONERA, 2 avenue Edouard Belin,
31055 Toulouse, France
[**] Atos Intégration SAS, 6 impasse Alice Guy, B.P. 43045,
31024 Toulouse cedex 03, France
{anthony.fernandespires,stephane.duprat}@atos.net
{thomas.polacsek}@onera.fr

Authors final version, accepted for publication as:

Anthony Fernandes Pires, Thomas Polacsek, and Stéphane Duprat. Formal software verification at model and at source code levels. In Alberto Abelló, Ladjel Bellatreche, and Boualem Benatallah, editors, Model and Data Engineering, volume 7602 of Lecture Notes in Computer Science, pages 162-169. Springer Berlin Heidelberg, 2012. 10.1007/978-3-642-33609-6_16.

The original publication is available at www.springerlink.com (http://link.springer.com/chapter/10.1007\%2F978-3-642-33609-6_16)

Abstract

In a software development cycle, it is often more than half of the development time that is dedicated to verification activities. Formal methods offer new possibilities for verification. In the specification phase, simulation or model-checking allow users to detect errors in models. In the implementation phase, analysis techniques, like static analysis, make the verification tasks more exhaustive and more automatic. In that context, we propose to take advantage of these methods to improve embedded software development processes based on the V-model.
Keywords: Verification, formal methods, development process, Model Based Engineering

1 Introduction

In software development, verification activities are significant costs. In the seventies, [8] was reporting that over half the software development time was devoted to tests. Today, in critical embedded software projects at Atos, we notice that the cost of verification activities can sometimes reach 60% of the total workload. In the DO-178b certification standard[1] verification means are reviews, analysis and tests, but the new version includes a specific text on the

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

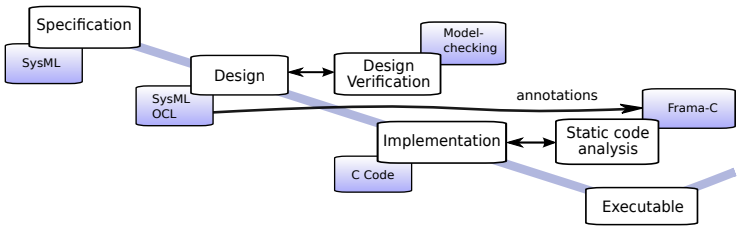

Figure 1: Left side of our V-model with verification tasks

use of formal methods (DO-333). Formal methods are mathematically-based techniques, for instance, formal logic, model checking or discrete mathematics. Although not all errors can be found with tests, most of them could be detected by the addition of formal methods in the early stages of development. It enables more effective identification of software defects and allows to reduce verification costs.

We propose to extend the embedded software development process based on V-model by introducing formal methods at the earliest stages of the life cycle and to use it to perform verification tasks. In this paper, our main contribution is to introduce links between model, formal verification and source code formal verification in an industrial context. Indeed, a lot of works focus on model verification and, more particularly, on OCL checking like we can see in [12] or [2]. Moreover, source code verification is successfully used now in industrial projects like [13]. But, all these works use verifications only in one step of the development cycle. Furthermore, because of our industrial context, it is necessary to consider the formalisms and the practices used by development teams. It is impossible to introduce a disruptive innovation. Consequently, we limit our work to OMG standards UML[2] and SysML[3] adapted to embedded software domain for the design stages and to Frama-C[4] for code analysis implementation stages.

In section 2, we introduce the addition of formal methods at the different phases of the left branch of the V-model. In sections 3, 4 and 5, we give a quick view of existing methods which can be used in the phases of our process, we describe what we do in our approach and we illustrate its use on a simple example. In section 6, we conclude the paper and give perspectives for our work.

2 The global picture

The V-model is the most widely development process used for embedded software. Even if it is often replaced, in other industrial domains, by other practices, it is not the case in our context. The V-model has two main streams: the development stream, the left side of the "V", represents the program refinement from requirements to code; the testing stream, the right side of the "V", represents integration of parts and their validation. In this paper, we propose to extend the left side of the "V" by adding earlier verification steps (figure 1).

[2] `http://www.uml.org/`
[3] `http://www.sysml.org/`
[4] `http://frama-c.com/`

In order to illustrate our approach, this paper uses a simple example of software development. We will represent the controller of a component in charge of the verification of the power. The controller is driven by a clock. At each clock tick, the controller does a task. During verification performed by the controller, if a problem is detected, the controlled component jumps to a maintenance status. While this maintenance status is maintained, no verifications are done by the controller. When the component status becomes normal, verifications start again.

3 Specification

3.1 Modeling Languages

There are many ways to specify software. The common one is to use natural language to write specifications, but it often leads to ambiguities and misunderstandings like explained in [9]. Modeling languages offer ways to produce formal specifications, which are better understandable, make communication easier between users and allow code and documents generation activities.

Our scope is UML/SysML standards. Some works adapt these modeling languages to the embedded domain. The UML profile MARTE [7] is an OMG standard for the modeling of embedded and real-time systems. It defines concepts in order to take into account the notions of time, concurrency, software and hardware platform, resources and characteristics like execution time. It is also possible to annotate models to perform analysis.

The SysML profile AVATAR [10] is dedicated to the modeling and the formal verification of real-time embedded software. The language is a specialization of SysML which focuses on activities realized in upper-stream of the development cycle. It offers solutions for: requirements engineering, system analysis, system modeling and safety and security properties modeling.

3.2 Our approach

We choose to use a subset of SysML for the specification. We limit the scope of elements and we define patterns for specific use, without adding new concepts. Our language is adapted to the specification of embedded software in a synchronous and scheduled environment. This kind of software is driven by a clock so it is expected to do a certain number of actions at each clock tick. Our subset is based on three diagrams: block diagram representing the structure of software components, state machine diagram representing the behavior of components and activity diagram, representing the detailed actions occurring in a state. The functioning is the following: for state machines, we constrain the triggers of all the major transitions with a unique event named “NextStep“ in our case. At each clock tick, which corresponds to one cycle, this event occurs. It allows the firing of a transition of the state machine. In this way, we control the evolution in a synchronous way. For more information and industrial use, see [6].

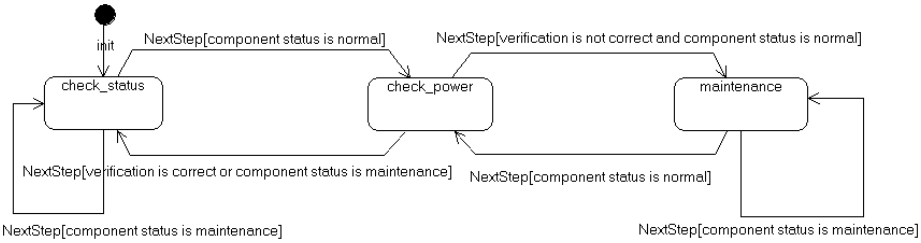

Figure 2: the controller specification

3.3 Application on the example

The behavior specification of the controller using the modeling language described above is given Figure 2. The controller starts by a state of verification of the component status (*check_status*). If it is normal, the controller goes to a state of power verification (*check_power*). If not, it repeats and waits for the next cycle to check the status again. Once in state *check_power*, the verification is proceeded. At the next cycle, if the verification is correct or if the component status is maintenance, the controller goes to *check_status* and repeats the behavior. If the verification is not correct and the component status is normal, the controller goes to a state of maintenance (*maintenance*) and the component status changes from normal to maintenance. The controller has to wait the end of the maintenance, which can last some cycles, before going to *check_power* again.

4 Design & verification

4.1 Formal models analysis

Model notations like those introduced in section 3, even though really efficient at conveying intelligible visual clues to the designers, lack the expressiveness needed to capture the finer details of a complete specification. For instance, expressing transition condition in natural language is a source of many ambiguities. To address this, formal constraints specification languages were introduced. The major interest of a fully formalized specification is the ability to perform analysis tasks in early phases of the development process.

Analysis tasks can be performed with model checking. Model checking, introduced in [11], allows users to verify if a system model respects a set of requirements expressed as properties. For example, in the embedded domain, UPPAAL [1] provides an environment for the modeling, simulation and verification of real-time embedded systems. The model checker can verify properties expressed in temporal logic Timed CTL, for which model checking is decidable. A gateway exists between UPPAAL and SysML. TTool [5] gives the possibility to check safety properties on AVATAR models with the UPPAAL model-checker.

[5] `http://ttool.telecom-paristech.fr/index.html`

4.2 Our approach

Here, we want to refine the specification model to meet a complete formalization of our specification. We translate guards in state machine diagrams from natural language to Object Constraint Language (OCL). OCL has been designed to be easy to read, to write and to understand. We can also detail for each state the set of actions that must be done in a cycle thanks activity diagram.

In addition, we want to use the formal specification of transition guards to check some simple properties. Our goal is to make the verification completely transparent to the user and to propose a set of properties to automatically verify thanks model checking. An example of property is the deadlock freeness of each state of the state machine. It is expressed with the following logical formula.

Definition 1 (Deadlock freeness of a state)**.**
Let s a state of a state machine and $G(s)$ the set of guards of all outgoing transitions of s where $G(s) = \{g_0, g_1, \ldots, g_n\}$. The state s is deadlock free if and only if $Dfree(s)$ is valid, with $Dfree(s) \equiv \bigvee_{0 \leq i \leq n} g_i$

4.3 Application on the example

In our example, we formalized each guard with an OCL expression. The result of the controller verification became a variable named *check_result* defined as enum type *checking_result* whose possible values are OK or KO. The component status became a global variable named *status*. This variable is of enum type *component_status* whose possible values are NORMAL or MAINTENANCE.

Today, we are not able to show the verification step, our work has just begun and the choice of the tool has not been yet decided. But these formalizations are sufficient to obtain code from the state machine and to conduct code analysis.

5 Implementation & static code analysis

5.1 Formal analysis of code

Static code analysis allows users to detect runtime errors or check properties on the source code without execution. The framework Frama-C is an open-source modular environment dedicated to the static analysis of C programs. The framework uses ACSL [6] specification language to specify properties and contracts on functions. Frama-C relies on different static analysis techniques available through plugins and linked solvers.

The value analysis technique, based on the abstract interpretation [3], can be used to ensure the absence of run-time errors in a program. This method is available with the Frama-C Value Analysis plugin [7]. It is based on the computation of variation domains for the variables of a program. The plugin gives warnings if it detects possible runtime errors, for instance access to invalid pointer.

Another method is the verification by proof obligations derived from the weakest precondition calculus introduced by [4]. It is a deductive method for

[6] ANSI/ISO C Specification Language, `http://frama-c.com/acsl.html`
[7] `http://frama-c.com/value.html`

proving properties. This kind of analysis is managed by Frama-C WP plugin [8] or Jessie plugin [9].

5.2 Our approach

In this step, our main goal is to ensure that source code conforms to the specification. In the same vein as [5], we want to derive ACSL annotations from the design model (here in SysML) to the code and check them with static analysis.

In our approach, the behavior of the software is represented by state machines. Each state machine is implemented by two C functions, a transition function dedicated to the choice of the triggered transition, the other for the behavior of each state. Each function is constructed as a *switch/case* structure. At each cycle, the two functions are called in sequence, in order to determine which transition is triggered and what will be the actions done in the cycle.

To verify the global behavior, we add a function contract for the C transition function. This contract is composed of one ACSL behavior for each state of the state machine. One behavior is composed of two types of clause. *Assumes* clause which specifies the property that must be true for the behavior to apply. In our context, it is the current state at the call of the function. Then, *ensures* clause which specifies the property that must be true at the end of the behavior. In our context, there is one *ensures* clause for each possible outgoing transition of the state. Each *ensures* clause is an implication defining that if the condition of the transition is true, it implies that the new state of the state machine is the targeted state of the transition.

In addition, we translate into ACSL the property expressed in design phase. We want to verify that a state, expressed in the model, is deadlock freeness using the enum types of the code. In that way, we use Frama-C to verify a property of the specification. For that we add annotations for each state in the C function. *Requires* clause will concern the range of the possible values of all variables used in the guards of all outgoing transitions of the state. The behavior, composed of one *assumes* clause will represent an implication defining that if we are in the current state, it implies our property (the translation of the logical expression defined in definition 1 to ASCL). Then, we will use a little trick. We will use the *complete behaviors* annotation, originally used to check the completeness of behaviors with Frama-C. In our case, this annotation will allow us to verify that the *requires* clause implies the *assumes* clause and so to prove our property.

5.3 Application on the example

The C code of the transition function of the example (see figure 2) is given below:

```
power_ctrl_state Power_Controller_T(power_ctrl_state state){
  power_ctrl_state o_state;
  switch(state) {
    case init :
      o_state=check_status;
      break;
```

[8] http://frama-c.com/wp.html
[9] http://krakatoa.lri.fr/jessie.html

```
    case check_status :
      if (status==NORMAL) o_state=check_power;
      if (status==MAINTENANCE) o_state=check_status;
      break;
    case check_power :
      if(check_result==KO && status==NORMAL) o_state=maintenance;
      if(check_result==OK || status==MAINTENANCE)
       o_state=check_status;
      break;
    case maintenance :
      if (status==NORMAL) o_state=check_power;
      if (status==MAINTENANCE) o_state=maintenance;
      break;
    }
  return o_state;
}
```

An example of ACSL behaviors is given below. This function contract checks the outgoing transitions of the state *check_power*.

```
/*@behavior state_check_power :
  assumes state==check_power;
  ensures (status==NORMAL && check_result==KO)
            ==> \result==maintenance;
  ensures (status==MAINTENANCE || check_result==OK)
            ==> \result==check_status;*/
```

The behavior representing the deadlock freeness for this state is given below.

```
/*@requires status==NORMAL || status==MAINTENANCE;
   requires check_result==OK || check_result==KO;
   behavior dfree_check_power :
   assumes state==check_power
             ==> ((check_result==KO && status==NORMAL)
                 || (check_result==OK || status==MAINTENANCE));

   complete behaviors dfree_check_power;*/
```

Static analysis is performed with the WP plugin version 0.4 of the Frama-C Nitrogen version. For our example, all the proof obligations are verified which means our transition function is compliant with our specification and the deadlock freeness is verified for each state of the state machine.

6 Conclusion

There are many formal methods for verification tasks and all of them can be incorporated in the different steps of the V-model. In this paper, we have presented how to use some of them throughout the development cycle to gain confidence on the specification, design and implementation, and to detect errors in early stage of the development cycle. In addition, we have begun to discuss how to link design and design verification with source code verification.

Furthermore, we have presented a way to verify a model property using code analysis tool.

This work is a first proposal. We still need to work on generation of ACSL annotations from SysML models. Today annotations are manually created from state machine models. We need to work on the consideration of activity behaviors and the automatic generation. We should also propose OCL patterns for properties and gateways to model checkers for the design verification step.

References

[1] G. Behrmann, A. David, and K. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 33–35. Springer Berlin / Heidelberg, 2004.

[2] J. Cabot, R. Clariso, and D. Riera. Verification of uml/ocl class diagrams using constraint programming. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society.

[3] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.

[4] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8:174–186, 1968.

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

[6] 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.

[7] S. Gérard, H. Espinoza, F. Terrier, and B. Selic. 6 modeling languages for real-time and embedded systems. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schtz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *LNCS*, pages 129–154. Springer Berlin / Heidelberg, 2011.

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

[9] B. Meyer. On formalism in specifications. *Software, IEEE*, 2(1):6–26, 1985.

[10] G. Pedroza, L. Apvrille, and D. Knorreck. Avatar: A sysml environment for the formal verification of safety and security properties. In *11th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*, pages 1–10, 2011.

[11] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[12] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying uml/ocl models using boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1341–1344, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[13] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *LNCS*, pages 532–546. Springer Berlin / Heidelberg, 2009.