Model Checking of SCADE Designed Systems

S Heim, Xavier Dumas, E Bonnafous, Philippe Dhaussy, C Teodorov, Lise Leroux

Model Checking of SCADE Designed Systems

S. Heim[1], X. Dumas[1], E. Bonnafous[1]

P. Dhaussy[2], C. Teodorov[2], L. Leroux[2]

1: CSSI, 3 rue du professeur Pierre Vellas, Toulouse, France
2: ENSTA-Bretagne, Lab-STICC UMR CNRS 6285, Brest, France

Abstract

Keywords: model checking, formal methods, CDL, SCADE, LUSTRE, OBP, synchronous, asynchronous

Introduction

Model checking **[1]** is a well-known method to verify a formal model in all possible configurations. Nevertheless this technique can hardly scale up to industrial **asynchronous** systems because of the **state-space explosion problem [17]**.

To address this challenge, a new approach based on **context specification** (the environment of the system) and an observation engine called **OBP** (Observer Based Prover) has been developed **[2]**. The idea is that given a property to be verified, one doesn't need to explore all possible configurations of the complete system. Among all **possible behavior** of the system, a tiny part is representative enough for the property to be verified.

Thus, specifying a pertinent environment (a context) allows **restricting the system behavior** on those only parts where the property is worth verifying.

The objective of our work is to apply this Context-aware verification method to the verification of **SCADE [3, 10]** systems designed in LUSTRE language, in order to check **behavioral properties** related to system safety.

Moreover **LUSTRE [4, 9]** is a **synchronous** language whereas OBP exploration engine takes as input an asynchronous model designed in **FIACRE [5]** language.

To cope with this problem our approach consists in developing a **GALS** method combining asynchronous contexts with synchronous models **[6, 7, 8]**.

The interest of our new approach is twofold:

- Verifying formal properties on synchronous industrial systems with formal methods using GALS approach,
- Facing the state-space explosion via context aware specification;

To our knowledge, there's no work combining those two previous methods.

This document is organized as follows:

First, a state of art on existing methods combining synchronous system modeling within an asynchronous environment is presented.

Next, we expose the **GALS** methodology approach we combined with context aware verification method.

Then we introduce two case studies used for experimentation of our method.

Eventually we conclude and present some perspectives for future work.

This work is done in the frame of the French R&D project **DEPARTS** [(1)], which is a **FSN/BGLE** project supported by **BPI France**.

1. State of the art

Mixing synchronous and asynchronous model designs. As demonstrated into previous studies (cf. Airbus **[12],** and Rockwell-Collins **[13]**), using an explicit model-checker for synchronous language verification may be required to verify some asynchronous properties between synchronous parts of a system.

Moreover, traditional "synchronous-observers" verification approach is not applicable in that case: asynchronous behaviors' modeling is not possible with synchronous language (i.e. communication delays, asynchronous clocks between processors).

GALS. Verifying a synchronous system in an asynchronous environment is not an easy task because of synchronicity assumption: Input/Output computations are considered to take no time.

To cope with this problem several GALS approaches have been developed **[6, 7, 8]**.

[(1)] DEsign PAtterns for Real Time and Safe applications.
FSN - Fonds national pour la Société Numérique.
BGLE - Briques Génériques du Logiciel Embarqué.

In **[7]**, the work consists in generating C code from the synchronous language **SIGNAL [14]**. Then, this code is called atomically (to ensure synchronicity assumption) in an asynchronous formal language: **PROMELA [15]**. The environment closing the overall system is then designed in PROMELA and the verification of a property is done with **SPIN** model checker **[15]**.

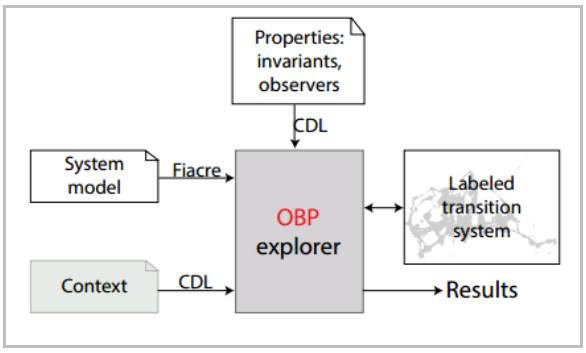
Nevertheless when dealing with huge systems, the environment grows drastically generating a state-space explosion. For this reason, an optimized exploration method has been developed based on OBP explorer and its associated Context Description Language called CDL [2].

Context-aware Verification approach. State-space explosion is intrinsically related on the way model checking method works. Model checking consists in **closing** a formal system with all possible behaviors of its environment, and then exhaustively analyzing the emerging executions. The idea behind the Context-aware Verification methodology **[2]** is that only a subset of the environment is necessary in accordance with the property one want to verify. This explicit description of the environment has many benefits:

- The environment can be decomposed by **several contexts** focusing on different system modes.
- When the environment is **too large**, it can be **decomposed** by OBP (splitting method **[11]**) to generate **independent sub-contexts, which are** successively composed with the system and the property so that to make several little verification.
- By enforcing some structural properties on the environment behaviors, OBP explorer can also use **optimized algorithms** such as PastFree[ze] to reduce the verification time **[16]**.
- Properties are verified only on specific context definitions.

The following picture could summarize the Context-aware Verification approach, implemented in the OBP toolkit.

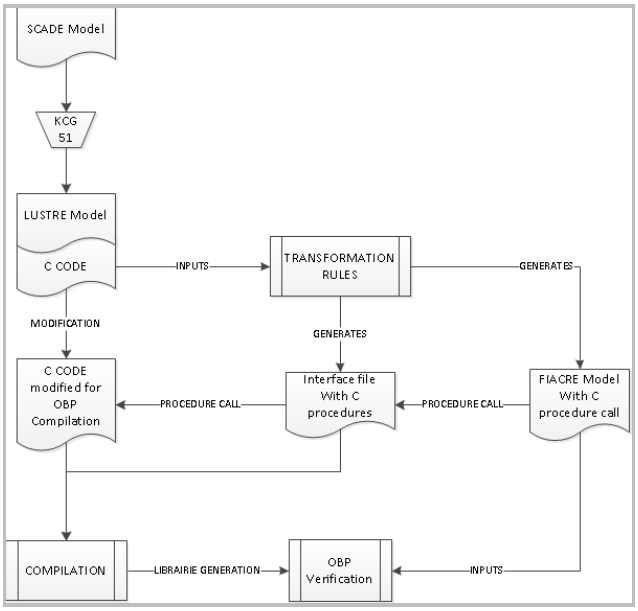
[2] www.obpcdl.org



2. Contribution

In this paper we propose to combine a GALS verification methodology with the Context-aware Verification approach. Our technique uses the synchronous LUSTRE language designed with SCADE tool and FIACRE asynchronous language used by OBP. We have experimentally validated our approach using two realistic case studies from the automotive and aero-space domain. In this study we focused on the verification of functional properties. Nevertheless our approach could integrate other classes of properties, and can accommodate techniques for guaranteeing the numerical accuracy.

The following picture summarizes our method.



Our approach is structured as follows:

1. We first design the system with SCADE components based on LUSTRE language;
2. Then we generate C code from the LUSTRE model thanks to the qualified SCADE code generator KCG51;

3. Next, from C code we generate the corresponding FIACRE model in accordance with the synchronicity assumption;
4. To make possible the compilation of C code, we generate some wrappers; the wrapper is useful to exchange data between FIACRE model and C code called functions;
5. FIACRE system with C code called function is generated so that to make the OBP exploration possible;

Once the FIACRE system is generated, following the previous five 5 steps one can implement an environment (an OBP context) to verify properties.

Nevertheless, some stimuli sent from the environment needs some parameters values so that to make the system under verification evolving.

To this purpose, we choose to generate a data structure into the FIACRE system containing all the input and output values which can be exchanged with the environment.

The data structure contained in the FIACRE model will be used for verification purpose by OBP tool.

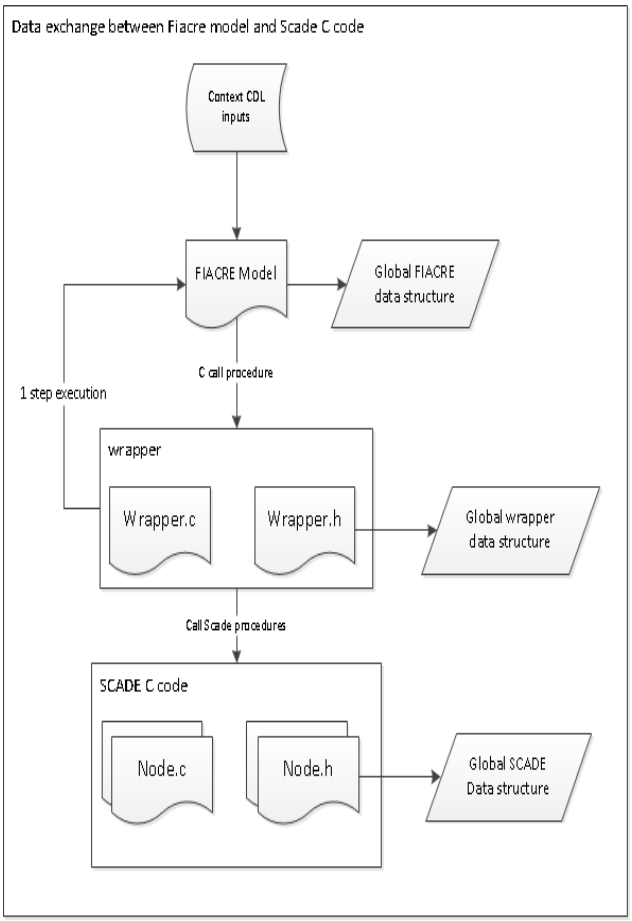
This implementation is well suited to our methodology because SCADE system stores all input and output values on global data structure too.

The stimuli and parameters sent by the environment are therefore copied into the FIACRE data structure (identical to SCADE C code data structure).

This data structure is passed in C call function parameters which are dedicated to the computations and modifications of input output values of the system.

When the C call function has ended, the data structure which has been modified is copied again into the FIACRE data structure so that OBP tool could display and verify properties from this FIACRE data structure.

The following picture summarizes the data exchanges between the environment, the FIACRE system and C call procedures.



3. Environment Modeling and Analysis

3.1 Environment modeling

In the case of Context-aware Verification, the environment modeling should be seen as a **methodological phase** that needs to balance two important constraints while building the context.

First the context has to cover enough behaviors to be considered valid for a given property. But at the same time it has to be small enough to be possible to exhaustively explore the product of its composition with the system under study (SUS).

The context is modelled starting from the **system requirements** one want to verify. From the requirements analysis, the designer identifies all the actors of the system that can interact and send some stimuli to the system.

For each actor, its behavior is refined by describing possible actions it can send to the system.

Eventually all the actors behaviors' are interleaved so that to generate all possible scenario of the environment.

Of course the more actors the worst, because each actor behavior is interleaved with all others potentially generating a combinatorial explosion of

the environment state space during unfolding and interleaving step.

This phase is done manually, and relies on the engineering judgment.

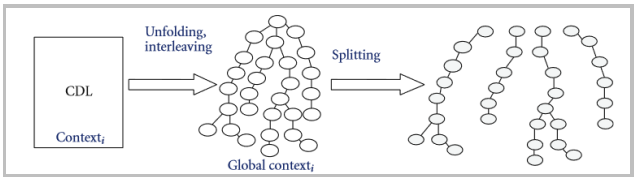
To face this environment state space explosion, a **pertinent modelling** of the environment must be described **by the engineer** who will for instance:

- discard some actors with no relation to the requirements,
- discard some useless actors stimuli with regard to the requirements he wants to verify,
- create several different environment with relation to the set of requirements he wants to verify,
- create a specific initialization sequence events to dig the system in a pertinent state for the verification.

3.2 Environment Analysis

Nevertheless a pertinent environment modeling is not always sufficient to face the environment combinatorial explosion.

For this purpose, the context aware verification tool (OBP) incorporates some algorithm to reduce environment state space explosion thanks to the splitting method.



The splitting method consists in decomposing the global context generated by the interleaving of all the actors' events in a set of global "sub-contexts" which will be composed with the system under study.

This method allows verifying systems stimulated by a complex environment, and covers exhaustively the whole generated state-space.

The combinatorial explosion environment behavior in space due to environment is transferred to combinatorial explosion in time due to the countless "sub-contexts" generated.

Nevertheless this new combinatorial explosion can be faced by parallel verification of the "sub-contexts" distributed to a set of machines.

An important observation is that while with the **automatic-split technique** the state-space is decomposed in several partitions, these partitions are not disjoint.

Hence the sum of these explorations with splitting represents the analysis of nearly two times more states and transitions than the exact initial state-space without splitting.

Nevertheless, we believe that this is a small price to pay for the possibility of analyzing five times larger state-space without the need of doubling the physical memory of the machine.

4. Case Studies

4.1 Roll-Control. We have first applied our method to a simple case study: a **Roll-Control** system.

The Roll-Control system allows to compute the Roll-Rate value, and to generate roll warnings whenever the roll rate is greater than 15° or lower than -15°.

The environment of this system is composed by three « actors »:

- Pilot actions on joystick
- Left and Right yaw applied on the plane

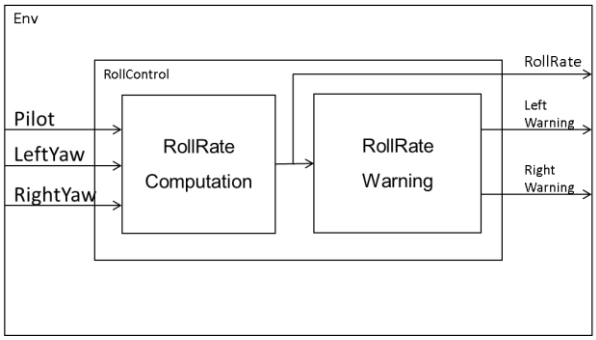
As a result of those inputs, the Roll-Rate is updated and warnings are activated in case the Roll-Rate is out of range.

First, the simplified coupling effect is calculated, then, the plane roll rate is calculated as follows:

```
rollCoupling = (leftAdvYaw – rightAdvYaw) × 0.1
rollRate = (joystickCmd – rollCoupling) × 0.25
```

The absolute value of the Roll Rate has to be saturated to 25.0.

The Roll-Control system is described in the following figure 4.1.



The Roll-Control is composed of 2500 lines of C code.

We have successfully checked following property on this model with OBP exploration engine:

The roll-control system shall never raise "left roll warning" and "right roll warning" at the same time;

This case study has successfully passed verification steps, because of its small size and limited possible behaviors.

4.2 Cruise-Control system. We then applied our method on an automotive Cruise-Control System (CCS) designed in SCADE.

This section provides an overview and some requirements of this case study.

Functional Overview. The CCS main function is to adjust the speed of a vehicle.

After powering the system on, the driver first has to capture a target speed, and then it is possible to engage the system. This target speed can be increased or decreased by 5 km/h with the tap of a button.

There are also several important safety features. The system shall disengage as soon as the driver hits the brake pedal or if the current vehicle speed (S) is out of bounds (40 < S < 180 km/h). In such case, it shall not engage again until the driver hits a "resume" button. If the driver presses the accelerator, the system shall pause itself until the pedal gets released.

Architecture overview (cf. Figure 4.2).

The CCS is composed of 3 mains parts: a “control panel”, a “system center”, and an “actuation manager” (for speed and throttle calculation).

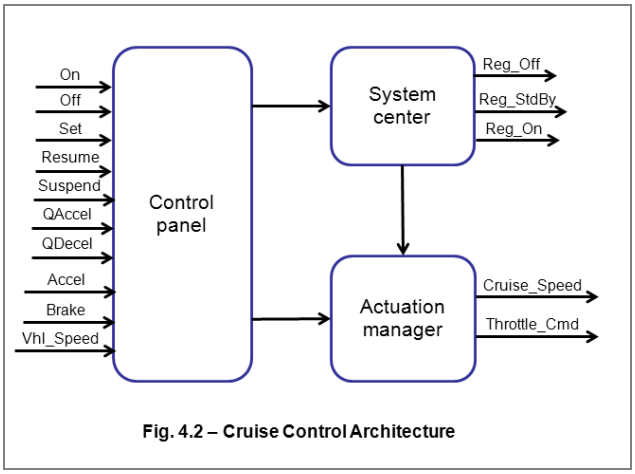


Fig. 4.2 – Cruise Control Architecture

The **control panel** is in charge of converting inputs signals from user to provide them to the system.

The **actuation manager** is able to capture the current speed and, once enabled, to adjust the vehicle speed to the defined target speed (and also throttle command value).

The **system center** component, that acts as a controller, and includes a state machine (states OFF, STDBY, ON).

The control panel acquires signals following from buttons used by the driver to operate the system:

- On, Off: Enable or disable the system
- Set: Capture the current speed as the target value
- Resume: Engage the control speed function
- Suspend: Disengage the control speed function
- QuickAccel: Increase the target speed by step
- QuickDecel: Decrease the target speed by step

In our case, the “control panel” is also responsible of providing BrakePressed and AccelPressed signals, which are built with Brake and Accel pedals signals, and to compute SpeedOutOfBounds signal. All are booleans signals provided to system center, with behaviours defined below:

- Brake pedal pressed: induces disengagement,
- Speed of the vehicle goes out of bounds: induces disengagement,
- Accelerator pedal pressed or released: pauses or resumes the speed control function;

The actuation module provides means for the system to interact with the vehicle. It can capture the current speed of the vehicle, and use it as a new target speed value. Once the CCS enabled, the actuation is responsible for controlling the vehicle speed accordingly.

Finally, the system center is the core of the CCS. It is responsible for handling events detected by control panel module.

Then system center use these events to switch to right system state, and to engage control function or not:

- From OFF to STD_BY: on btn_On,
- From STD_BY to ON: on btn_Set,
- From ON to STD_BY: on btn_Suspend or Brake,
- From STD_BY to OFF: on btn_Off;

Requirements. This section lists three main requirements of the CCS system and shows how to model them using the CDL formalism, with predicates or observers automatons.

REQ.1: The system shall not engage itself if the target cruise speed is not set.

REQ.2: The target Speed shall never be lower than 40 km/h or higher than 180 km/h.

REQ.3: When the system is powered off, the target speed shall be reset, and considered as unset.

REQ.1 can be encoded by using an **observer automaton**, on figure 4.3 below. To encode this observer using CDL formalism, we first need to introduce the events triggering the transitions

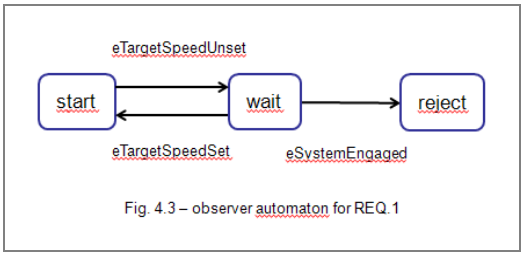


Fig. 4.3 – observer automaton for REQ.1

```
4.4  predicate pCruiseSpeedIsUnset is {
     {sys}1:context._Cruise_speed < 40
     or {sys}1:context._Cruise_speed >180 }
     event eTargetSpeedUnset is {
     pCruiseSpeedIsUnset becomes true }
```

On listing 4.4, pCruiseSpeedIsUnset is a predicate on cruise speed value, read from interface structure (context) of the main process {sys}, returning true if the constraint is verified.

Then an event eTargetSpeedUnset is built with "becomes true" formula in order to express a rising edge of the predicate, which is an **observable event** in OBP observation engine.

On listing 4.5, another event can be defined based on system center state machine output value (Regul_ON).

```
4.5  predicate pSystemIsEngaged is {
     {sys}1:context._Regul_ON = 1 }
     event eSystemEngaged is {
     pSystemIsEngaged becomes true }
```

Using these events, the observer automaton of figure is defined in listing 4.6:

```
4.6  property REQ1 is {
     start   -- eTargetSpeedUnset  --> wait;
     wait    -- eSystemEngaged     --> reject;
     wait    -- eTargetSpeedSet    --> start }
```

We can then encode two others requirements REQ.2 and REQ.3 using the same principles.

To model predicates and events, we could also use internal states of concurrent processes of the system.

Environment. In the case of the CCS the environment is built from two main distinct actors modeling:

a) a nominal scenario,

b) a disruptor;

The basic scenario can be seen as a linear use case of the CCS that covers all the functionality involved by the properties we aim to verify.

This scenario must pass through following steps:

Event	Behavior
Press Accelerate pedal	Vehicle speed grows
Button On	CCS ON / stand-by
Button Set	Target speed set
Stop Accelerate pedal	CCS engaged / regulate
Brake pedal	CCS in stand-by
Resume button	CCS engaged
Button Off	CCS OFF

The disruptor is a wide alternative including changes of the vehicle speed within the allowed range or not, pressure on the pedals and the panel buttons. The disruptor stresses the SUS against a number of possible unexpected behaviors of the environment.

The disruptor encoding refers to verification environment capability of sending events to system, including speed target requests, pressure on pedals, or on panel buttons.

```
4.7  activity disruptor is {
     eRegularSpeed_v1/v2
     [] eAbnormalSpeed_v3/v4
     [] ePressPedals_p1/p2
     [] ePushButton_b1/b2/b3 }
```

Once these two actors are composed asynchronously, we get a wide range of variations of the basic scenario using the capabilities of the disruptor at all stages.

```
4.8  Cdl myContext is {
     Properties req1
     Assert req2, req3
     Init is { eBtnOn }
     Main is { basic_scenario
               || disruptor }
     }
```

4.3 Verification Results

This section presents the results obtained for the verification of the main requirements previously presented, emphasizing the importance of the Context-aware Verification approach, applied on our GALS approach.

During the exploration, we have tried several kinds of observers, and we have intentionally create errors into the model in order to check that there were detected (assertion, or reject state of automaton).

The verification results for the CCS case study are:

- 17.847 states and 62.771 transitions (with 3 processes, in 21 sec);
- 367.800 states and 1.621.000 transitions (with 4 processes, in 571 sec)

So we can conclude that our approach does not produce a too large state space, due to encapsulation technique used (atomic execution of synchronous function into asynchronous process).

Even if we have not yet modeled all required behaviors as asynchronous communication between several more synchronous processes, it seems to be a very promising approach for larger and complex systems.

As a comparison, our partner from **Lab-STICC** has applied same verification context on an asynchronous CCS UML model, which generates 3 millions of states and 10 millions of transitions (with 4 processes, and 4 ticks of clock only) (cf. **[18]**).

For the moment, we have only used traditional Breadth-First Search (BFS) reachability algorithms. But we know that in case of a larger state space, we could also use PastFree[ze] algorithm and splitting technique.

The use of the PastFree[ze] algorithm enable the analysis of a 2.4 times larger state-space, and the joint use of PastFree[ze] and automatic split technique enable 4.78 times larger state-space, compared to traditional Breadth-First Search (BFS) reachability algorithms, without the need of extending the physical memory of the machine.

5. Conclusion and Perspectives

In this paper we have used the Context-aware Verification technique for the analysis of several requirements of a Cruise-Control System composed of synchronous languages functions.

Modeling and verification of asynchronous properties of this kind of systems based on composition of synchronous components, renders traditional synchronous model-checking approaches inefficient.

Using the environment reification through the CDL formalism, this task becomes manageable by relying on two powerful optimization strategies.
These strategies rely on the structural properties of the CDL contexts and enable the reachability analysis of larger industrial models.

While the approach presented in this paper offers promising results, for this technique to be used on industrial-scale critical systems, some work has to be done on the formalization of the context coverage, with respect to the full-system behavior, in order to assist the user on initial context specification.

References

[1] Clarke, E.M., Emerson, E.A., Sistla, A.P.: "Automatic verification of finite-state concurrent

systems using temporal logic specifications", ACM Trans. Program. Lang. Syst., vol. 8, p. 244--263, ACM, New York USA, 1986.

[2] Dhaussy P., Boniol F., Roger J.C.: Reducing state explosion with context modeling for model-checking. In: 13th IEEE International High Assurance Systems. Engineering Symposium (Hase'11). Boca Raton, USA (2011)

[3] “SCADE 6: A Model Based Solution For Safety Critical Software Development”, François-Xavier Dormoy. ERTS 2008, Esterel Technologies;

[4] “The Synchronous Dataflow Programming Language Lustre”, N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, Proceedings of the IEEE, 79(9):1305-1320, September 1991.

[5] B. Berthomieu, J. P. Bodeveix, M. Filali, H. Garavel, F. Lang, F. Peres, R. Saad, J. Stoecker, F. Vernadat. The syntax and semantics of FIACRE v2, specification, 2009.

[6] H. Garavel and D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In Proc. of SPIN, pages 241–260, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] F. Doucet, M. Menarini, I. H. Kruger, R. Gupta, and J. P. Talpin. A Verification Approach for GALS Integration of Synchronous Components. Electron. Notes Theor. Comput. Sci., 146:105–131, January 2006.

[8] H. Günther, S. Milius, O. Möller: On the formal verification of systems of synchronous software components, VerSyKo project, 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP, MBEES), 2012.

[9] “The Foundations of Esterel”, Gérard Berry. In “Proofs, Languages, and Interaction, Essays in Honour of R. Milner,” G. Plotkin, C. Stirling, and M. Tofteed., MIT Press (2000).

[10] Methodology Handbook Efficient Development of Safe Avionics Software with DO-178B Objectives Using SCADE, 2011;

[11] Dhaussy, P., Boniol, F., Roger, J.C., Leroux, L.: Improving model checking with context modelling. Advances in Software Engineering ID 547157, 13 pages (2012)

[12] “Model-checking Flight Control Systems : the Airbus experience”, ICSE Companion, pages 18–27, T.Bochot (Airbus), V.Wiels (ONERA), P. Virelizier, H. Waeselynck, 2009.

[13] Choi, Y., “From NuSMV to SPIN: Experiences with model checking flight guidance systems”, Formal Methods in System Design 30 (FMSD) , 199-216 (2007)

[14] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. Proceedings of the IEEE, 79(9): 1321-1336, September 1991.

[15] Holzmann, G. J., The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2004. ISBN 0-321-22862-6.

[16] C.Teodorov, L.Leroux, P.Dhaussy: Context-aware Verification of a Cruise-Control System, In Proceedings of Model and Data Engineering 4th International Conference, MEDI 2014, LNCS 8748;

[17] Valmari, A.: The state explosion problem, in Springer LNCS volume 1491, 1998, pp 429-528 (http://dx.doi.org/10.1007/3-540-65306-6_21)

[18] P. Dhaussy, L. Le Roux, et C. Teodorov : Vérification Formelle de Propriétés, application au cas d’étude CCS en UML, dans le cadre du projet DEPARTS, Revue Génie Logiciel, n° 109, juin 2014