Debugging Embedded Systems Requirements before the Design Begins *

*** "The beginning is the most important part of the work" - Plato**

Fabien Gaucher and Yves Génevaux, ARGOSIM SA
8-10, rue de Mayencin, 38400 St-Martin d'Hères, France
{Fabien.Gaucher|Yves.Genevaux}@argosim.com

Abstract

Over the last two decades, there has been a real movement towards requirements engineering [1]**. However, the various tools that have been developed with the aim of improving the specifications development are mostly focused on requirements management and traceability (Doors, Reqtify, etc), and there is no practical upstream requirements validation tool available for checking their functional coherence before the design stage. As a result, over half of the faults detected during the testing phases result from specification mistakes** [2]**. This is particularly prejudicial in the case of critical embedded systems where specifications play a central role in the certification process.**

Argosim ambition is to change this situation by providing STIMULUS [3], the very first requirements simulation tool for real-time embedded systems. STIMULUS combines the ability to formalize requirements into a text-based language which is both simple and expandable, but also and even more importantly, to simulate them by generating the possible behaviors of the specified system in the form of execution plots. The analysis of simulation results allows for the quick detection of incorrect, incomplete or even conflicting requirements. In practice, STIMULUS integrates seamlessly into the agile development processes [4], by allowing for the concurrent development of both the requirements and their test scenarios in a very incremental way. Test scenarios thus developed will make it possible to generate many test vectors that will be usable at the design validation stage, while the requirements will be reused as such in order to specify and verify test objectives.

This paper describes the major innovation offered by the STIMULUS tool in the field of requirements engineering. It will present its operating principles which will be illustrated with a specification item relating to the automobile domain, starting with the definition and the development of the requirements through their simulation up to the embedded code validation tests against those requirements.

Keywords

Requirements Engineering, Textual Specifications, Simulation, Constraint Solving, Debugging, Validation, Automatic Testing, Real-time Embedded Systems.

I. The Requirements Engineering Challenge

Validating the specifications for embedded systems is a twofold challenge, especially in the case of critical applications. On one hand because any specification error results in design errors, whose correction will require costly iterations of the development process. On the other hand, because those systems are subject to safety standards that impose high quality criteria on the requirements [5]. Today, the functional validation of requirements relies on the three following approaches:

- The manual review of the specifications documents, which does not allow for the detection of any likely ambiguities nor subtle inconsistencies among the various specifications at stake.
- The use of syntactic or even semantic analysis tools [6], that help to create the requirements in a more formatted and structured format; again, despite such approaches allow to detect some mistakes, they do not provide any support to the system's functional validation.
- The running of code validation tests once the system is implemented, which makes it possible to challenge the specifications, thus very late in the development cycle.

We can also mention formal verification tools [7] but they are intended for use exclusively by verification experts, not by system architects, and in practice, their scope of industrial application is restricted by both decidability and complexity issues.

So why being still here in 2015, whereas the "model-based system engineering" approach appears everywhere and the use of simulation processes in systems design activities has stood out as an evidence over the last 20 years?

The reason for that is that contrary to the system designer, the system architect is expected to focus on the "what", namely on *what* the system is supposed to do and not on the "how", that is how to precisely execute the specified function [3]. In order not to interfere with the design process, the specification activity must avoid as much as possible to describe the implementation choices, in order words it must reduce the options available to those in charge of the implementation.

And that is where the simulation problem comes in: generally speaking, the term *simulation* refers to the execution of an *algorithm*, detailing step by step every data computation. The execution is therefore deterministic since, in other words, it has to do with implementing "how" the system will perform.

Since it should be avoided to describe in detail those implementation choices during the specification process, there is nothing to simulate with existing tools. Hence the lasting blocking situation that has prevailed for 20 years, in which on one hand many tools and techniques have been developed and become largely widespread for the design, coding and test phases, while on the other hand the specification phase remains limited to the reference tool: a text editor, typically MS Word.

The result of such *status quo* is that 40 to 60% of the errors which are detected at the testing stage result from ambiguous, incomplete, erroneous or conflicting specifications. How to get out of this dilemma and its highly costly consequences for companies engaged in the development of complex systems?

II. STIMULUS Operating Principles

Following a number of fundamental research results in the field of formal languages and validation methods, notably carried on in the work of the VERIMAG laboratory [8] [9], Argosim released STIMULUS, a highly innovating tool which specifically addresses the challenge described above. How? By applying the two following basic principles:

1- STIMULUS considers requirements as constraints imposed on the specified system behavior. A constraints solver [10] is therefore able to compute the space of possible behaviors at a given execution cycle. A random generator can then produce a wide range of possible executions of the specified system without needing neither a design model nor a computer code.

2- In order to make STIMULUS easily available to requirements practitioners who are used to express requirements in natural language, the requirements are formatted in text using the "boiler plates" methodology [11]. Since each textual template is also formally defined by means of constraints, STIMULUS makes it possible to write requirements that look very similar to their original formulation, while being executable.

The edition of formalized textual requirements is a key asset of the tool. **Fehler! Verweisquelle konnte nicht gefunden werden.** gives an example of a simple requirement written in STIMULUS.

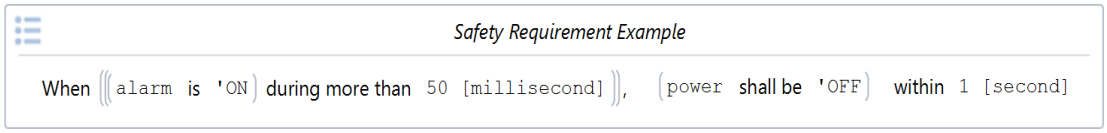

Figure 1 - Display of a formalized textual requirement in STIMULUS

This requirement was edited by "dragging and dropping" several predefined templates, namely *When*, *DuringMore* and *Within*. The set of predefined templates is not fixed and the user may build his own domain-specific templates by composing existing ones or raw constraints, and by associating it to some natural language sentence with "gaps".

In addition to the textual form, STIMULUS also provides modeling features like stochastic state machines and blocks diagrams. The first will allow to describe operational modes of the system, while block diagrams will support a functional or architectural decomposition, with clear interfaces presenting the signals exchanged between sub-systems. The blocks hierarchy will also ease the refinement of high level requirements into low level requirements as well as the allocation of those latter requirements to the various sub-systems.

This approach makes it possible to bring simulation at the specification level, providing unprecedented debugging capabilities to perform early and effective validation of functional real-time requirements.

III. Validation of System Requirements

In order to illustrate how, in practice, STIMULUS makes things easy for system architects to formalize and debug requirements, this section will focus on some simple function from the automotive industry, namely a car headlights control system.

One of the high-level requirements for the automatic mode is to prevent the headlights from flashing. In order to formalize such a requirement in STIMULUS, we will define a new template stating that a signal does not flash if, *after* an instability, the signal remains stable *during* at least 2 seconds.

Associating a sentence format with this new operator will make it possible to use it as a template within many requirements. This new operator may be textually formatted as "_ *is not flashing*", cf. **Fehler! Verweisquelle konnte nicht gefunden werden.**, where the gap "_" will be replaced by any signal name or expression.

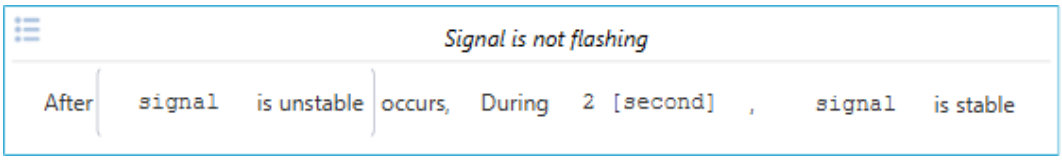

Figure 2 - Composing existing templates to define a new operator

The high-level requirement will then be formalized by applying the new "flashing" operator to the "*headLight*" signal, as in Figure 3.

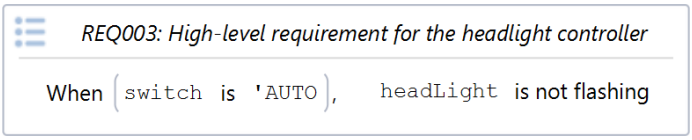

Figure 3 - Using a new operator inside a requirement

The resulting requirement is both readable and non ambiguous, since it can be executed with a perfectly defined semantics.
It is possible to refine this high-level requirement REQ_003 by specifying more precisely the lighting controller behavior with low-level requirements describing some hysteresis behavior:

- **REQ_003.1:** if the switch is turned to AUTO, and the light intensity is at or below 70% then the headlights should stay or turn immediately ON. Afterwards the headlights should continue to stay ON in AUTO as long as the light intensity is not above 70%
- **REQ_003.2:** if the switch is turned to AUTO, and the light intensity is above 70% then the headlights should stay or turn immediately to OFF. Afterwards the headlights should continue to stay OFF in AUTO as long as the light intensity is not below 60%
- **REQ_003.3:** if the switch is in position AUTO, the headlights are OFF, and the light intensity falls below 60%, then the headlights should turn ON if this condition lasts for 2s
- **REQ_003.4:** if the switch is in position AUTO, the headlights are ON, and the light intensity is above 70%, then the headlights should turn OFF if this condition lasts for 3s

Upon careful reading of these requirements, one may take it for granted that the hysteresis behavior has been correctly specified and will prevent any flashing of the lights. We are going to demonstrate that this may not be the case.
These four requirements can be specified with predefined templates from the STIMULUS standard library, which will offer a textual formulation close to the natural language. For instance, Figure 4 gives the STIMULUS formulation for the REQ_003.1 requirement.

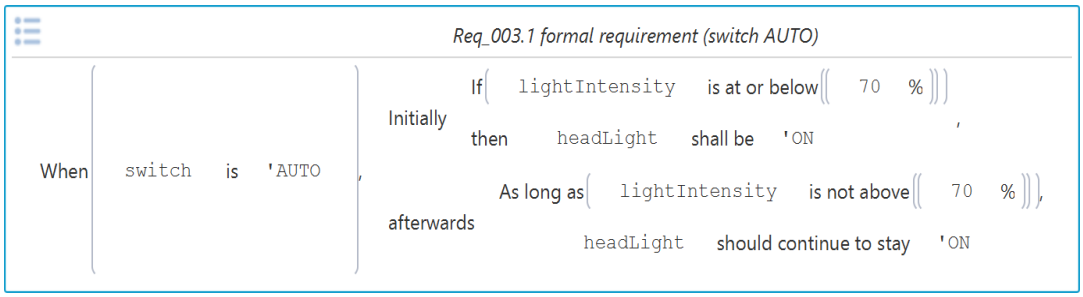

Figure 4 – Requirement REQ_003.1 formalized in STIMULUS

But since STIMULUS allows to generate different possible executions of the specified system, how does the lighting controller behave within such formulation? Are the low-level requirements describing some hysteresis behavior that is compatible with the high-level requirement stating that the lights should not flash?

Before simulating requirements, we can specify basic hypotheses for the light intensity behavior as well as the automatic mode command. In other words, we can describe a test scenario for the requirement. If no starting scenario is provided, STIMULUS will generate random data by default for the system's input signals. As recommended in safety standards, we propose to describe a test scenario in STIMULUS that will for instance induce light luminosity fluctuations between 55% and 75%, corresponding to the interesting fluctuation range in automatic mode.

The test scenarios are connected to the low-level requirements (block REQ_All) through the block diagram of Figure 5, which in turn relates to the high-level requirement specified in parallel. STIMULUS will simulate the whole of it and try to induce behaviors satisfying both low-level and high-level requirements as well as the test scenarios.

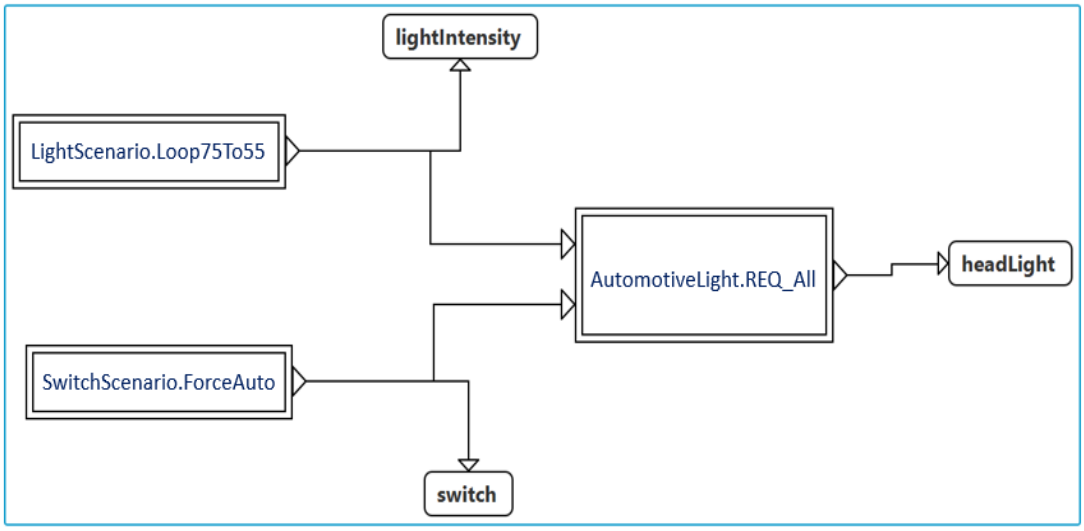

Figure 5 – Connecting system requirements to test scenarios

The test scenarios are also specified in the form of textual constraints, allowing to generate an equivalence class of test vectors. For instance, the Loop75To55 scenario induces a light intensity fluctuation between 75 and 55%, with a dedicated "*goes up and down*" operator whose definition only determines the light intensity limits and the alternated up and down phases, but not the light intensity variation amplitude, cf. Figure 6. As for the ForceAuto scenario, it simply forces the automatic mode by assigning the switch value to AUTO.

```
lightIntensity  goes up and down respectively to  75  % and  55  %

switch  is set to  'AUTO
```

Figure 6 – A test scenario specified as constraints on system inputs

The simulation of all the requirements, including the block diagram with low-level requirements and test scenarios, as well as the high-level requirement, will produce the plot of Figure 7.

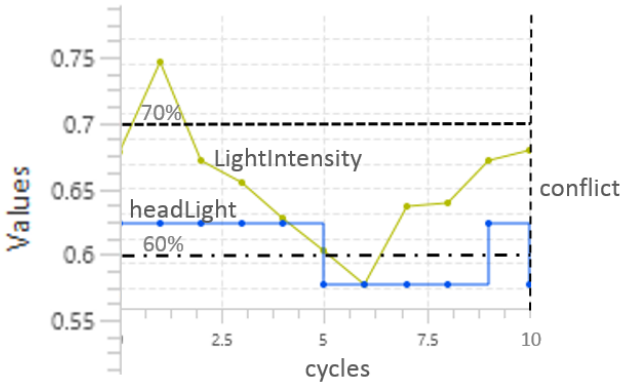

Figure 7 – A possible execution of the specified system

After 10 cycles, STIMULUS automatically detects a requirement conflict and highlights the various requirements responsible for the conflict. In this particular case, there is a violation of the high-level requirement (the headlights should not flash) by a low-level requirement. Each time STIMULUS reports a conflict, it also identifies automatically the minimal set of conflicting requirements.

Then, the simulator's debugging features (error messages, highlight of active and inactive requirements, access to the various signal values at any time of the execution, etc) allows users to thoroughly analyze the conflict, identify its cause in order to modify the faulty requirements or add some missing ones.

Once the faults are corrected (in this particular case, the use of the AsLongAs operator within the REQ_003.1 and REQ003.2 requirements should be replaced by a When operator), one gets the expected behavior, and the absence of conflict indicates that the low-level requirements now satisfy the high-level requirement, see Figure 8.

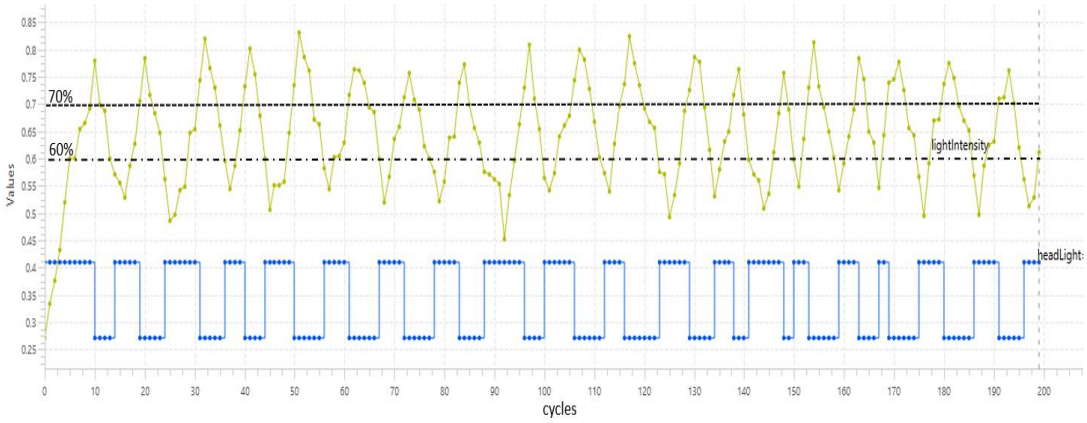

Figure 8 – Simulation results with the corrected headlights requirements

Here, STIMULUS made it very quickly possible to:

- Formalize the requirements and generate behaviors for the specified system;
- Detect a conflict between requirements (in this case between the high- and low-level requirements);
- Correct faulty requirements and validate them through some iterative simulation process.

IV. Validation of the System Implementation

Until now, we've shown how to use STIMULUS to validate functional requirements before any design or coding takes place. We are now going to illustrate in this section how to reuse the various requirements and test scenarios in order to validate the system implementation, once it is available.

The principle is to:

- Substitute the executable requirements with the system code to be tested inside the block diagram. For that purpose, we need to define some external system in STIMULUS that will embed the compiled code of the system implementation and define the same interface (inputs/outputs) as the system

of executable requirements. When simulating this new block diagram, the compiled code will be stimulated by the same test scenarios as the requirements.

- Check that the system implementation behaves as specified in the requirements by turning the requirements into observers, or test oracles. STIMULUS will then automatically report any requirement violation occurring during simulation. In practice, one simply need to drag&drop any system of requirements beside the block diagram and turn it into monitoring mode, represented by a light blue eye icon, as illustrated in Figure 9.

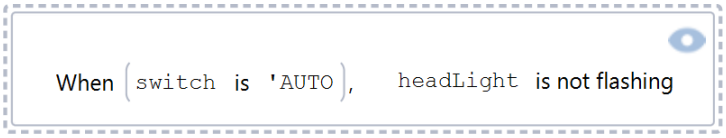

Figure 9 – the hig-level requirement used as a test observer

For the purpose of our demonstration, we tested a system implementation written in C. We used the block diagram to connect the C code with the previously defined scenario in which:

- The switch is set to AUTO;
- The light intensity is between 55 and 75%.

The STIMULUS simulator produces results similar to those of Figure 8 but this time, *headLigh*t is produced by the compiled code which replaces the executable requirements.
The STIMULUS interface makes it possible to monitor requirements violations panels, in which the green color and the "true" state indicate that the requirements are constantly satisfied by the compiled code, whereas the red color and the "false" state indicates a requirement violation. The left-hand side panel of Figure 10 reports that no requirement violation has been observed during the whole simulation when using the previous test scenario.

Figure 10 – Observer panels: no requirement violation on the left, one requirement violation on the right

In order to validate the system under a wide range of fast mode and luminosity changes, the constraints of the previous scenario may be relieved, so as to let

STIMULUS generate less likely situations and combinations that might have been omitted.

It is for instance possible to define a second scenario in which:

- The switch may change at any time to ON, OFF or AUTO;
- The light variation is simply restricted to + or - 10% per cycle.

After 200 cycles, the STIMULUS simulator now reports at least one violation for the REQ_003.1, signaled by the red color and the state "false" inside the right panel of Figure 10. The precise time step at which the requirement has been violated can be observed inside the simulation plot by selecting the requirement in the monitoring panel. The requirement monitoring diagram displays the problem soon after the 175th cycle, as shown in Figure 11.

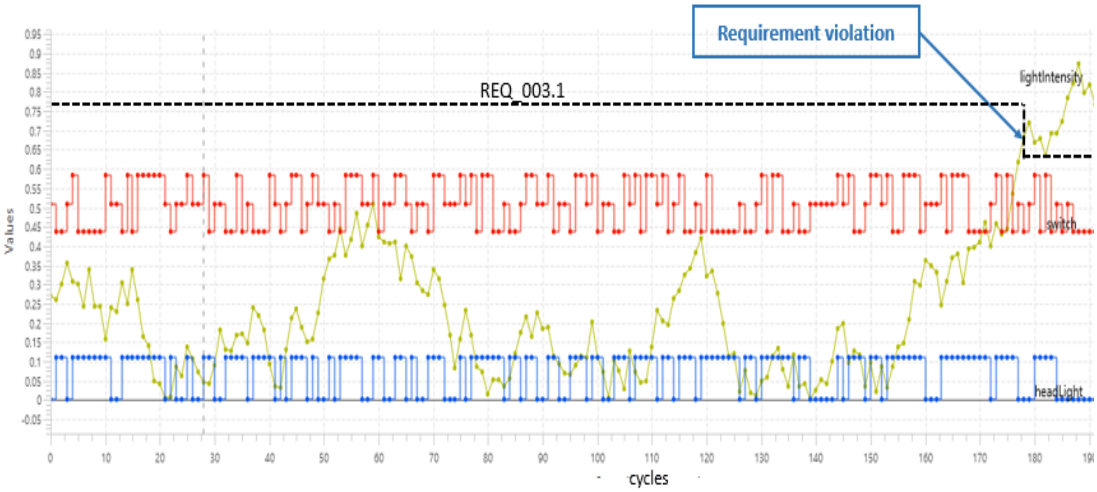

Figure 11 – Software-in-the-loop simulation tests with a requirement violation detection

By zooming in for a closer look, it appears that the problem occurred:

- Just after the light intensity reached the hysteresis zone, within 60 and 70%;
- Just after the OFF *switch* turned to AUTO.

In this particular setting, the code generates the OFF value for *headLight*, while REQ003.1 required the value ON for *headLight*. This bug symptom can then be analyzed to find its cause in the C code. The test case revealing the fault may be exported into the non-regression tests database.

STIMULUS generated this problematic test case by automatically exploring a large number of control modes and mode changes, as well as various light intensity ranges and variations. Even with such a limited case, the combination range would already be wide enough for a manual testing not to explore all interesting scenarios. With STIMULUS, the process induces little work since, instead of describing specific individual test cases, it relies on test cases "classes" defined by constraints. Of course, STIMULUS does not guarantee a complete exploration of the state-space, but it provides coverage information on both the scenarios and the requirements.

At last, the test objectives are not derived any more from the requirements since they *are* the requirements themselves. In other words, the system implementation is directly validated against the original requirement formulation, which has been formalized and already validated during the requirements specifications process.

V. Conclusion

In this paper, we have presented STIMULUS, the first commercial tool that brings simulation capabilities at the specification level to allow for the early validation of functional embedded system requirements. This tool offers totally new and effective features for system architects to make requirements right the first time and validation engineers to check that the system implementation satisfies its requirements.

The ability to formalize textual requirements which are close to the natural language, while being perfectly defined, allows for development teams and stakeholders to share non ambiguous requirements that can fit to the specific application domain a well as to the culture and usages of the company.

This textual language is also useful for describing non deterministic test scenarios that will generate a class of test vectors, in other words, the tool will explore an envelope of the test scenario (different variations and combinations of signals within the possible behaviors).

When defined, debugged and validated together during the specification process, both requirements and test scenarios can be reused to test the system implementation. Since this test environment is available before the design begins, it provides a great support to agile processes, where the implementation code can be validated in a very automatic and incremental way: test vectors can be generated automatically each time a new implementation is available and requirements can be checked automatically through requirements observers.

In terms of scalability, STIMULUS provides a natural way of validating system requirements by “layers”. At a given system level, requirements are defined with respect to inputs/outputs. If this system level is made of different sub-systems, these requirements can be refined into low-level requirements that will be allocated to sub-systems, and so on, following the block diagram hierarchy.

Future work includes the use of reachability analysis, when applicable, in order to guide simulation traces towards interesting points, for debugging purpose, or in order to explore rare paths and improve test coverage of requirements and scenarios.

References

[1] K. Pohl, Requirements Engineering: Fundamentals, Principles, and Techniques, Springer Publishing Company, Incorporated, 2010.

[2] G. J. Myers and C. Sandler, The Art of Software Testing, John Wiley & Sons, 2004.

[3] B. Jeannet, Debugging Real-Time Systems Requirements : Simulate The “What” Before The "How", Nürnberg: Embedded World Conference, 2015.

[4] P. Abrahamsson, J. Warsta, M. T. Siponen and J. Ronkainen, New directions on agile methods: a comparative analysis, Washington, USA: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003.

[5] IEEE 29148 - Systems and software engineering -- Life cycle processes -- Requirements engineering, IEEE STANDARDS, 2011.

[6] A. Fraga, J. Llorens, L. Alonso and J. Fuentes, Ontology-Assisted Systems Engineering Process with Focus in the Requirements Engineering Process, Fifth International Conference on Complex Systems Design & Management (CSDM), 2014.

[7] S. P. Miller, A. C. Tribble, M. W. Whalen and M. P. E. Heimdahl, Proving the shalls: Early validaiton of requirements through formal methods, International Journal of Software Tools for Technology Transfer, 2006.

[8] P. Raymond, Y. Roux and E. Jahier, Lutin: A language for Specifying and Executing Reactive Scenarios, EURASIP Journal on Embedded Systems, 2008.

[9] E. Jahier, N. Halbwachs and P. Raymond, Engineering Functional Requirements of Reactive Systems using Synchronous Languages, Porto, Portugal: International Symposium on Industrial Embedded Systems (SIES), IEEE, 2013.

[10] P. Schrammel and B. Jeannet, Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs, Static Analysis Symposium (SAS), 2011.

[11] A. Mavin and al., EARS (Easy Approach to Requirements Syntax), 17th IEEE International Requirements Engineering Conference, 2009.