Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study

Bertrand Jeannet, Fabien Gaucher

Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study

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

Abstract—**In a typical software project, 40% to 60% of design bugs are caused by faulty requirements that generate costly iterations of the development process as specifications need to be redefined, design and implementation modified accordingly, and then retested. The major reason for this situation is that no practical tool exists for debugging requirements while drafting specification, and the many tools that exist for requirement management and traceability do not address this problem.**

STIMULUS provides an innovative solution for the early debugging and validation of functional real-time systems requirements. It provides a high-level language to express textual yet formal requirements, and a solver-driven simulation engine to generate and analyze execution traces that satisfy requirements. Visualizing what systems will do enables system architects to discover ambiguous, incorrect, missing or conflicting requirements before the design begins.

We demonstrate the use of STIMULUS on the specification of automatic headlights from the automotive industry. We show how this unique simulation technique enables to discover and to fix ambiguous and conflicting requirements, resulting in a clear and executable specification that can be shared among engineers.

Keywords—Requirement Engineering, Real-time Embedded Systems, Domain Specific Languages, Formal Methods, Debugging, Simulation.

I. INTRODUCTION

Many tools have been proposed for the development of embedded software, in which the validation activity may represent more than 60% of the whole development effort. In this process, functional validation aims at checking that the system design is correct with respect to requirements, but few tools exist for the functional validation of system requirements themselves.

In this paper, we focus on real-time requirements, such as the following cruise control example:

"When active, the cruise control shall not permit actual and desired speeds to differ by more than 2 km/h during more than 3 seconds."

Such requirements usually describe a combination of logical and numerical properties of system signals over time. They stand in contrast to non-functional requirements, such as performance, usability, reliability, cost, etc. Being requirements, they express what a system should do or not do, but they do *not* describe how to achieve it: here for instance it could be done with a PID (proportional-integral-derivative) controller.

In practice, requirements are mostly written in natural language and are generally validated through manual reviews. As a consequence, many ambiguities and errors remain until validation testing. It is well-known that the later these errors are detected, the more expensive it is to fix the bug. The cost is even worse when third parties are involved as the extra process iterations involve specification and contractual changes.

Argosim STIMULUS addresses this issue of requirement early debugging and validation by providing two key features:

(I) Expresses real-time requirements and environment assumptions in a formal yet close to natural specification language;

(II) Generates and observes simulation results that satisfy requirements under environment assumptions.

The ability to formalize requirements in an easy-to-read language is a necessary condition for the approach to get acceptance by users, while the ability to simulate "what systems shall do" makes requirements validation possible while writing specifications, instead of delaying it to a later phase of the development process. This limits specification errors and ultimately reduces costs in the design phase.

In this paper we show how STIMULUS supports these claims by describing its technical foundations and by illustrating its use to formalize, debug and validate the requirements of a car automatic headlights controller which was provided to us by a Japanese software and tools vendor.

II. Related Work

Several requirement engineering tools do exist, such as IBM DOORS. Compared to STIMULUS, they focus on requirement management and traceability, rather than validation.

Specification and simulation tools, like UML/SysML or Mathworks Simulink, aim at modeling and validating system design and architecture rather than high-level requirements. There are efficient at describing *how* a system should be implemented, but they lack the expressiveness to describe and to simulate *what* a system should do without describing the *how*. The case-study of this paper will clarify this point.

Formal methods tools, in particular model-checkers and proof systems, like the Rodin platform based on Event-B [1], provide expressive languages and exhaustive validation features. There are the tools to use to *prove* the consistency of a set of worked-out requirements, but less so to incrementally *debug* partial, possibly incorrect requirements and to *discover* missing requirements, for the two following reasons.

1. They can only detect formalized inconsistencies: they do not help the user to discover problems that he has not anticipated.

   Consider the cruise control example given in introduction. Before submitting this requirement to a model-checker or a proof assistant, the user has to define a relevant property to prove on it, which is not trivial. Moreover, there is no *a priori* guarantee that the chosen property can detect a mistake like omitting the absolute value operator when translating the condition "speeds should not differ by more than two km/h". In contrast, as we will see, the simulation feature of STIMULUS enables the user to discover unanticipated problems in requirements by observing simulation traces.

2. Model-checkers and proof assistants require fairly complete requirements to issue relevant results, that is, either a successfull proof of a non-trivial property on them, or a meaningful counter-example to it. This makes this approach hard to use in an incremental specification process where rough initial requirements are progressively refined.

The two approaches are actually complementary:

- STIMULUS enables to debug requirements and to validate them by simulation; its strength is its ability to quickly exhibit problems. This provides a level of confidence in the quality of requirements, which is arguably higher than the level provided by manual reviews, but do not deliver either a formal proof of consistency and correctness.

- Once good quality requirements are obtained, they can be submitted to model-checkers and/or proof systems to obtain formal proofs. Problems can still be discovered at this step, but much less frequently than if the previous step had not been performed.

This complementarity motivated a partnership with SafeRiver, a consulting company specialized in safety and cyber-security for software-based systems. SafeRiver makes intensive use of formal methods and tools, and is looking for solutions to speed up the correct formalization of safety requirements before performing exhaustive proofs on them.

III. STIMULUS Technical Foundations

The scientific backgrounds of STIMULUS being detailed in [1], we give only an overview of it. The two key features of STIMULUS are an expressive formal specification language and a simulation engine based on a constraint solver.

A. A Constraint Real-Time Specification Language

Stimulus combines the concepts of the synchronous languages LucidSynchrone [5] and Lutin [6]:

- LucidSynchrone (like its industrial version SCADE) provides proven and mature concepts for modeling real-time systems, such as dataflow equations, hierarchical state machines, and synchronous parallel composition.
- Lutin provides the concepts needed for modeling real-time *non-deterministic* behaviours, namely dataflow constraints and non-deterministic control choices. It was designed for describing generic test scenarios for real-time systems.

We list below the main concepts of the resulting language.

a. Data: Synchronous Data-Flow Constraints

The behaviour of signals is specified with dataflow constraints like:

```
count = (0 -> last count) + (if evt then 1 else 0);
count <= 10;
```

in which the integer signal `count` counts the number of occurences (in time) of the Boolean signal `evt` and is constrained to be less than or equal to 10. `->` is the initialization operator: `e1 -> e2` evaluates to `e1` on reset, and to `e2` otherwise, and `last v` denotes the value of signal `v` in the previous step. In other words, this specifies than `evt` might be true at most ten times during an execution, but specifies nothing more about when `evt` can or cannot be true.

b. Control: Hierarchical State Machines.

They are typically used to model running modes, to define the temporal operators of the standard library, and also to model probabilistic choices in scenarios. States can contain constraints, as shown by the state machine of Fig 1. State machines in STIMULUS have a strong, preemptive transition semantics according to the terminology of [5]: transitions are fired and their condition evaluated in the same step as their destination state. In addition, they implement an original notion of termination which happens to be crucial to combine the temporal operators of the standard library.

c. Modularity: Systems and Macros.

As other synchronous languages or Mathworks Simulink, Stimulus encapsulates statements into *systems* that can then be

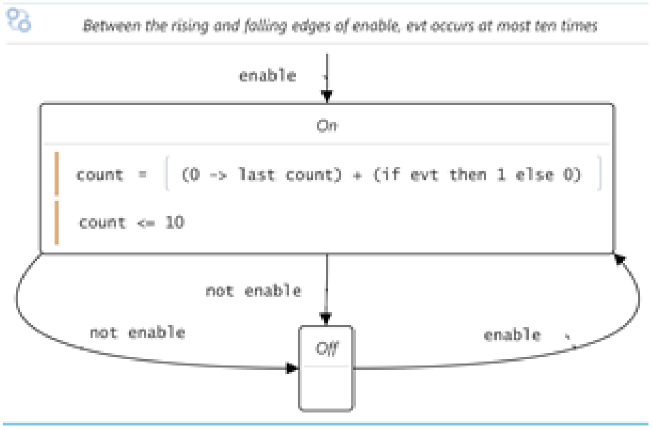

Between the rising and falling edges of enable, evt occurs at most ten times
enable
On
count = (0 -> last count) + (if evt then 1 else 0)
count <= 10
not enable
not enable
Off
enable

Fig. 1: requirement about the occurrences of evt

instantiated at several places. One can for instance define a system *Count* that counts the number of occurrences of an event (using an internal memory) and that can be reused to specify more complex constraints, like `Count(evt1) >= Count(evt2)`.

Stimulus also provides a robust system of *macros* with clear scoping and typing rules that accepts statements as parameters, in addition to signals having a value. They are used to define temporal operator, such as the

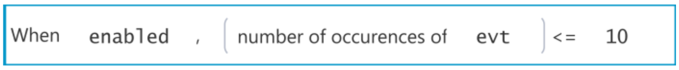
when <*condition*>, <*BODY*>

operator defined by the macro depicted on Fig. 2.

d. Readability: Sentence Templates

Formal requirements are easier to read when they look like textual requirements instead of programs. To achieve this goal, a system or macro can be associated with a user-defined *format* that specifies how instances should be edited and displayed in the editor.

For instance, if one associates to the macro *When* and the system *Count* the formats `when %condition%, %<BODY>%` and `number occurences of %event%`, the requirement of Fig. 1 rewritten using When and Count will appear as:



When enabled , number of occurences of evt <= 10

which is arguably more concise and more readable than the automaton of Fig. 1, while still enjoying the same unambiguous, formal definition.

e. Architecture: Block Diagrams.

Systems can be instantiated not only as a sentence, as described above, but also as blocks connected to other blocks in block diagrams, like in Mathworks Simulink, see Fig. 8. This enables developers to graphically describe the architecture of a system and to visualize the flow of information in it.

Other technical features of the STIMULUS language are a construct for controlling constraint propagation and orienting constraints, in the spirit of [12], which provides scalability both for solving constraints and for model understanding by the user, type inference techniques of functional languages [13,14] to minimize the annotations required from users, and a physical dimension analysis [15,16] to statically detect this kind of inconsistencies.

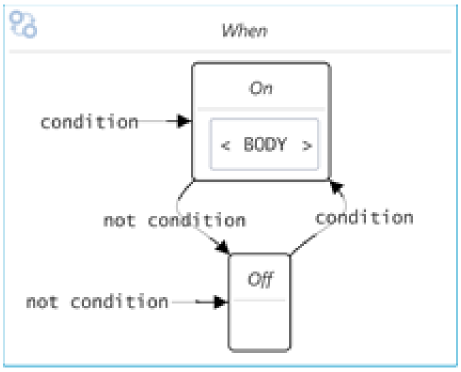


Fig.2 Macro defining the temporal operator *When*

Regarding the modeling of time, currently STIMULUS provides a simple periodic physical time model, by using a period to assign physical time values to logical time steps. In the long term we plan to provide a more flexible, aperiodic time model.

B. Compilation Process and Simulation Engine

SIMULUS compilation process follows the principles of [3,4,5,6] for (i) reducing parallel composition of hierarchical state machines to sets of statements guarded by choices on the *clocks* representing the active states of state machines, and (ii) ordering statements properly with respect to dependency constraints.

The simulation engine follows the principles of [7,8] and combines an exploration algorithm for resolving the non-deterministic choices induced by state machines and a constraint solver for resolving the non-determinism induced by constraints on variables. The solver handles logico-numerical constraints mixing logical operators on Boolean and enumerated variables, and linear constraints on numerical variables. The restriction to *linear* numerical expressions concerns only the unknown variables to be solved: the solver can deal with non-linear sub-expressions on variables that are known at solving time, like inputs or memories. For instance the constraint `x*(last x) >= y*(last y)*(last y)` is linear on the unknowns `x` and `y`.

IV. METHODLOGY FOR DEBUGGING REQUIREMENTS

In the previous section, we gave an overview of a new real-time constraint programming language built by combining the two research lines of work lead by M. Pouzet [3,4,5,6] and by the synchronous team of VERIMAG laboratory [7,8,9,10]. Now, how can it be used to debug requirements of real-time systems ?

With the current practice, requirements are mostly textual that are worked out and validated through manual reviews, and then given as inputs to

1. system designers and programmers on the one hand, who will implement the system;
2. test engineers on the other hand, who are in charge of writing functional test cases and test verdicts to confront the implementation w.r.t. the initial requirements.

Fig. 3 depicts the functional test bench architecture of a real-time system. I and O denote resp. the inputs and outputs of the System Under Test (SUT), which is considered as a black box. The box "Scenarios" feeds the SUT with inputs I, possibly taking into account outputs O to produce realistic inputs. The box "Requirements" reads both inputs I and outputs O of the IUT and emits a verdict.

As mentioned in the introduction, the experience shows that half of the bugs discovered by functional tests are requirements bugs, and not implementation bugs. The problem is that these bugs can be found only after the SUT becomes available.

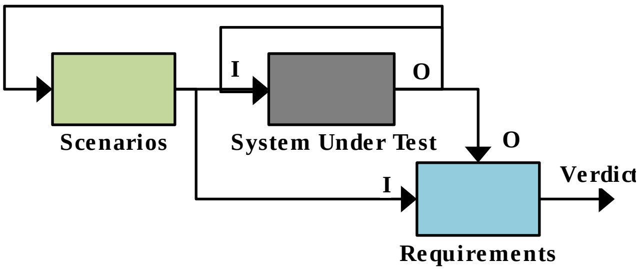


Fig. 3 : Classical test architecture of a real-time system

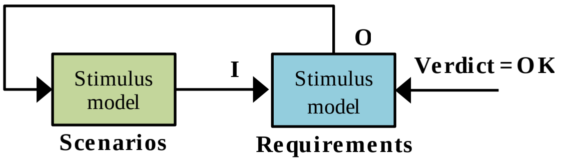


Fig. 4 : Debugging architecture for requirements and scenarios with STIMULUS

In contrast, the methodology made possible by the STIMULUS constrained-based language and its simulation engine is the following one:

- Requirements are seen as constraints between inputs I, outputs O, and the verdict (OK/NOK).
- The box "requirements" in Fig. 4 act as an outputs *generator*. At each simulation step:
    - the inputs I and the verdict OK are provided to the box "requirements";
    - the simulation engine solves the constraints in the remaining unknowns O and picks a random solution for them.
- Similarly, scenarios are also seen as constraints between I and O, which may describe general assumptions on inputs and/or more specific test cases.
- The box "scenarios" in Fig. 4 acts as an inputs generator: given outputs O provided to it by the feedback loop, the simulation engine solves the constraints in the remaining unknowns I and picks a random solution for them.

The fact that scenarios may be generic scenarios with a large variability on control and data allows the simulation engine to generate many different simulation traces, and ultimately makes more likely the discovering of an unexpected behaviour corresponding to a problem in the requirements. Actually scenarios are optional: one may simulate requirements alone. However, in practice, one often needs some general assumptions on the stability or the variation of signals to make simulation traces readable or realistic.

Another important observation is that once scenarios and requirements have been worked out using the architecture of Fig. 4, they can be reused directly in the test architecture of Fig. 3 with a black-box system under test: indeed, there are executable, and STIMULUS have a mechanism to turn a generator into an observer, which enables to turn the requirements box of Fig. 4 in the requirements box of Fig. 3.

We described in the two previous sections the scientific foundations of STIMULUS and the methodology it enables for debugging requirements together with their associated generic test scenarios. The sequel of the paper aims at answering to the following questions:

- How does it work in practice? Is it easy to formalize informal textual requirements with STIMULUS? How far is the formalized version to its informal counterpart in term of readability (traceability feature)?
- Are simulation traces effective at discovering problems? In other words, despite the non-exhaustiveness of our validation-by-simulation approach, is it "exhaustive enough" in practice?

To answer to these questions, we will illustrate the use of STIMULUS for formalizing and debugging requirements of an automatic light system coming from the automotive industry.

V. Formalizing the Requirements of Automatic Headlights

These requirements were provided to us by a Japanese software and tools distributor, as a typical example of the kind of requirements his customers have to deal with in the automotive industry. We insist on the fact that they have not been specifically invented for evaluating a tool like STIMULUS, neither by us nor by researchers or developers of alternative solutions.

The original, textual specification of the automatic headlights is depicted on Fig. 5. The head sentence is a sort of informal, high-level requirement that describe the general purpose of the four more precise, lower level requirements that follow. This purpose is to command the switching of the lights. In the sequel we will formalize the four requirements named **3Aa, 3Ab, 3B** and **3C**.

Fig. 6 depicts the initial formalization of requirement **3Aa,** which will be debugged by simulation and upgraded in the

If the switch is AUTO then the headlights turn on or off, depending on the ambient light intensity - with a defined hysteresis to prevent blinking.

REQ_003Aa: 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%.

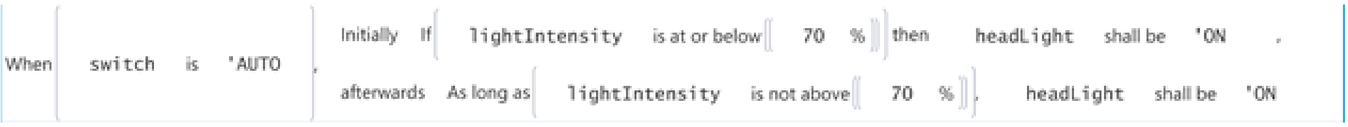
REQ_003Ab: if the switch is turned to AUTO, and the light intensity is above 70% then the headlights should stay or turn immediately OFF.

Afterwards the headlights should continue to stay OFF in AUTO as long as the light intensity is not below 60%.

REQ_003B: if the switch is in position AUTO, the headlights are OFF, and the light intensity falls bellow 60%, then the lights should turn ON if this condition lasts for 2s.

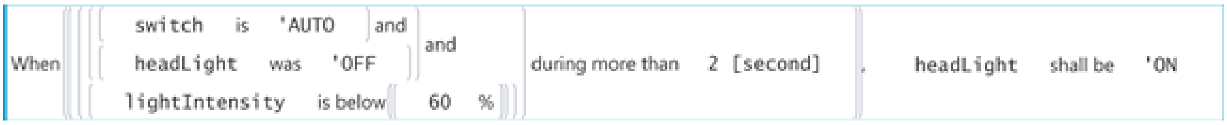
REQ_003C: if the switch is in position AUTO, the headlights are ON, and the light intensity is above 70%, then the lights should turn OFF if this condition lasts for 3s.

Fig. 5: Original specification of the automatic headlights



When switch is 'AUTO , Initially If lightIntensity is at or below 70 % then headLight shall be 'ON . afterwards As long as lightIntensity is not above 70 % , headLight shall be 'ON

Fig. 6 : Requirement **3Aa** in STIMULUS, initial version



When switch is 'AUTO and headLight was 'OFF and lightIntensity is below 60 % during more than 2 [second] , headLight shall be 'ON

Fig. 7 : Requirement **3B** in STIMULUS, initial version

next sections. One recognizes in Fig. 6 the pieces of sentence underlined on Fig. 5. This formalization is based on the use of STIMULUS standard library, which provides a number of sentence templates that are ubiquitous in real-time requirements, such as:

- "**as long as** <*expression*>**,** <*statement*>"
- "**initially** <*statement*>, **afterwards** <*statement*>"

As explained in Section III.A.d, such sentence templates are user-definable views for formal systems and macros, and their purpose is to make easy tracing the formalized requirement back to the textual, non-formalized requirement.

The other requirements **3Ab**, **3B** and **3C** are similarly formalized. **3Aa** and **3Ab** roughly specifies an hysteresis. **3B** and **3C** in addition require the light intensity to be low or high to hold for some time before switching respectively on or off the headlights. This is to prevent the headlights to blink too quickly, for instance when driving under a bridge.

VI. Simulating and Refining the Requirements

We just showed how the requirements can be formalized. However the major innovation of STIMULUS is the ability to simulate them.

A. *Simulation architecture*

Fig. 8 depicts the block diagram defining the simulation architecure considered in this paper. The block **Env** generates values for the signal **lightIntensity** that satisfy the assumptions depicted on Fig. 9. These assumptions combine general physical assumptions on the range of the light intensity (a percentage of a maximum intensity) and of its derivative, and a scenario making it alternatively increase or decrease. In this paper, we maintain the signal **switch** to **AUTO**, as we want to simulate the behaviour of the system under this mode.

The block **R003_v1** on Fig. 8 is defined by the requirements discussed in Section **Erreur ! Source du renvoi introuvable.**. This block will generate possible values for **headLight** that are compatible with these requirements and the values of the other signals.

B. *First simulation and detection of a problem*

Fig. 10 depicts a possible execution trace of the system defined by Fig. 8. One can observe the behaviour of **lightIntensity** and **headLight** signals. L60 and L70 are the two thresholds 60% and 70% that appear in the requirements.

What can be observed is that at start **headLight** has the expected behaviour: it is first OFF because the light intensity is above the 70% threshold, and when the light intensity falls below the 60% it becomes ON. However, afterwards the behaviour seems completely random and appears in contradiction with the intended behaviour.

Hence the simulation exhibits a problem either in the textual requirements, or in their formalization, or in both.

C. *Investigating and solving problem 1*

Consider the textual requirement **3Aa** on Fig. 6 and more precisely its second part:

"[…] Afterwards the headlights should continue to stay ON in AUTO **as long as** the light intensity is not above 70%".

The textual expression "as long as" is actually *ambiguous*: does it mean

1. **as long as** condition, something [**afterwards nothing**]" (sequential behaviour)

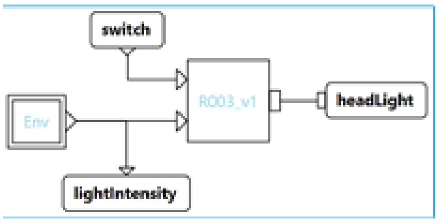
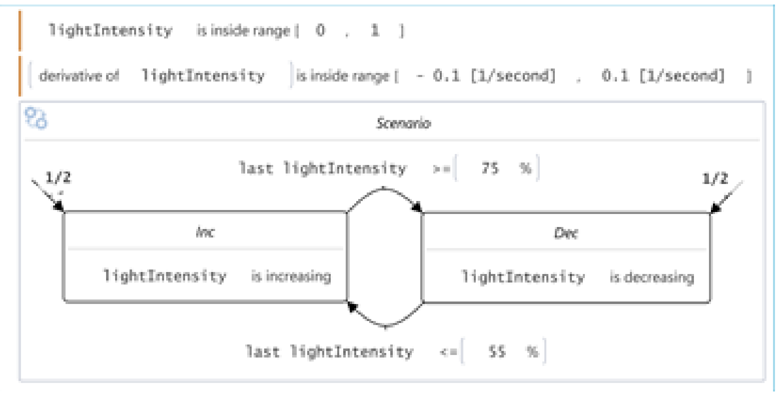


Fig. 8: simulation architecture



lightIntensity is inside range [0 , 1]

derivative of lightIntensity is inside range [- 0.1 [1/second] , 0.1 [1/second]]

Scenario
last lightIntensity >= 75 %
1/2
1/2
Inc
lightIntensity is increasing
Dec
lightIntensity is decreasing
last lightIntensity <= 55 %

Fig. 9: system *Env* describing the assumptions on the light intensity

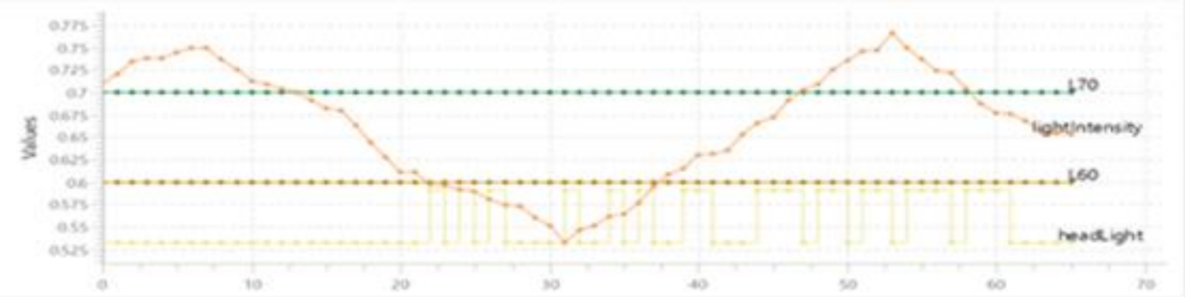
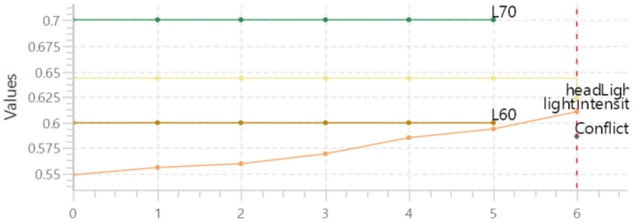


Fig. 10: Simulation of requirements, initial version

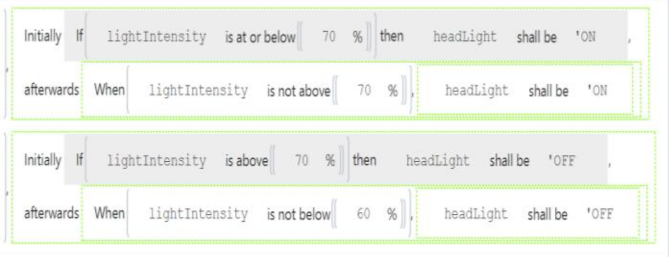
2. or "[**always**] **when** condition, something" (cyclic behaviour)?

This ambiguity is not really an artefact introduced by STIMULUS sentence templates: it is a real ambiguity which already exists in the textual version, between a sequential or a cyclic behaviour.

On Fig. 6 we opted inadvertently for the first interpretation, but clearly we expect a cylic behaviour here. Let us try the second one, which is available in the standard library. Requirement **3Ab** which follows the same pattern is similarly modified. Let us simulate this new version of requirements:



We obtain a *conflict* at simulation step 6: this means that some requirements are contradicting each other. Let us highlight the requirement **3Aa** and **3Ab** in the debugger at step 6 where the conflict occurs:



Initially If lightIntensity is at or below 70 % then headLight shall be 'ON ,
afterwards When lightIntensity is not above 70 % , headLight shall be 'ON

Initially If lightIntensity is above 70 % then headLight shall be 'OFF ,
afterwards When lightIntensity is not below 60 % , headLight shall be 'OFF

The debugger highlights the active parts of the requirements, and allows the user to discover that at this step, the requirements implie **headLight** to be ON and OFF at the same time!

D. *Investigating and solving problem 2*

Consider again the textual requirement **3Aa** on Fig. 1. We formalized the expression

"headLights should continue to stay ON"

underlined in Fig. 1 with the formalized sentence

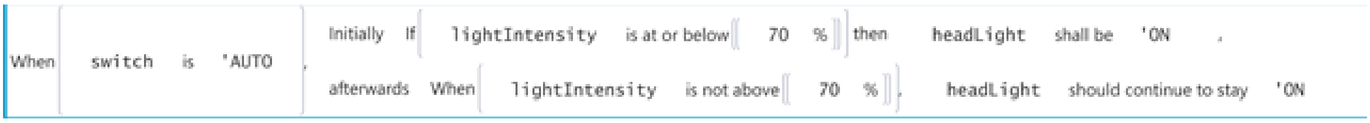
"headLights shall be `ON`".

Was that the right interpretation? The sentence could also mean:

"if it was ON, maintain it at ON, otherwise do nothing".

To check this hypothesis, we defined a new, user-defined sentence template

"*<expression>* **should continue to stay** *<constant>*"



When switch is 'AUTO , Initially If lightIntensity is at or below 70 % then headLight shall be 'ON .
afterwards When lightIntensity is not above 70 % . headLight should continue to stay 'ON

Fig. 11 : Requirement **3Aa** in STIMULUS, second correction

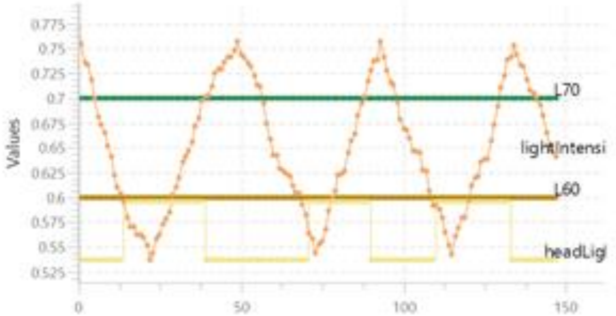


Fig. 12: Simulation of requirements after the second correction

with this semantics and used it to update requirement **3Aa,** resulting in the requirement of Fig. 11. Requirement **3Ab** was similarly modified.

Fig. 12 depicts a possible execution trace with this new version of requirements, which follows much better the expected behaviour of the head lights which should not blink and which should switch to ON or OFF according to two thresholds.

A last problem, that we will not detail as much, concerns the requirements **3B** and **3C**: they express that something should happen when a condition hold for some time duration, but say nothing about what should happen before. Hence unwanted, blinking head lights behaviour might occur with a more generic scenario in which the switch is not always equal to AUTO. This is a typical example of a missing requirement.

VII. DISCUSSION

The four textual requirements of Fig. 5 look rather simple and reasonable. Yet they contain several ambiguities and omissions, some of them leading to unwanted behaviours, others to contradictions. Actually, although looking simple, they specify a complex behaviour that depends on the past history of signals, and on physical time. This complexity behind apparent simplicity is typical of real-time requirements and explains the strong need for their early validation, especially as they are often requirements of security-critical systems.

A. Comparison of simulation approach to alternative requirements validation approaches

It is difficult to discover by manual reviews the problems we discovered by simulation. This is why many of them are discovered later, either by developers or test engineers when they start exploiting the requirements given to them, or, worse, even later during the functional test phase.

We claim that model-checker or formal proof systems are not very suitable either for this *debugging and elicitation* task. For detecting and fixing the first problem found in Section VI.D, it is necessary

1. to explicitly formalize the higher-level property capturing the bad behaviour, which requires having already identified it as a potential error;
2. to check the validity of the requirements against this property – this will fail of course;
3. to analyse the cause of the failure and to try a new alternative.

The simulation approach made possible by STIMULUS

(i) completely removes the need for subtask 1; the user can still insert property observer to automatize the detection of an already identified potential problem, but this is optional;
this subtask is actually replaced by the observation of simulation traces enabling the *discovery* of unanticipated behaviours or getting an higher confidence in the relevance of the requirements

(ii) makes subtask 2 arguably easier: model-checkers may have an issue with too complex models, either in terms of size or expressiveness (non-linear computations for instance), and proof systems are not always fully automatic; in contrast simulation techniques are less computationally demanding and can handle complex models fully automatically[1].

Subtask 3 remains the same but in overall the removal or the simplification of the other subtasks make the trial-error cycle much quicker.

The use of model-checking and formal proof approaches is of course still meaningful to obtain an exhaustive confirmation of consistency, or to discover inconsistencies occurring in very uncommon cases that a simulation approach may miss.

B. Evaluation w.r.t. announced criteria

At the end of Section IV describing our methodology, we proposed two main evaluation criteria to our approach, namely:

1. Easiness of formalization, readability of formal models and traceability w.r.t. informal, textual requirement;
2. Effectiveness of the simulation engine in generating simulation traces exhibiting problems, that is, sufficient "practical exhaustiveness".

[1] As mentioned in Section III.B, the constraints of STIMULUS models should be linear, but only on the unknown variables at solving time, which is not restrictive in practice.

We hoped to have successfully convinced the reader that the formalized requirements of Figs. 6 and 7 remain close to and as concise as their informal counterpart of Fig. 5. Although these formal requirements are not pure English, everybody can understand what they are talking about. In addition, they gain a perfectly formal semantics: the user can look at the formal definitions of the sentence templates to get a more in-depth understanding of a sentence, and simulate them to observe their dynamic behaviours.

Technically, this is a benefit of the combination of a powerful programming language (which is mostly hidden to the user), the use of sentence template as a view for programming constructs, and the design of well-thought standard library.

Regarding the second evaluation criteria, the two problems discovered in Section VI were both discovered with the very first simulation trace. This means that although the validation-by-simulation approach is not exhaustive, in practice the simulation engine is efficient at quickly exhibiting problems when they exist. In particular, the conflict discovered at the end of Section VI.C, which would have been easily found by a model-checker, was in practice as easily found by the simulation engine, using the very unspecific generic scenario of Fig. 9.

Of course, if the assumptions on the environment disallow some scenario, STIMULUS will not exhibit problems that are specific to them. But the same applies to model-checkers and formal proof systems: formal proofs are valid only w.r.t. the considered assumptions on the environment. STIMULUS actually supports the incremental process of starting debugging requirements with very simple scenarios, and later stimulating them with more complex or corner-case scenario. This process allows the user to progressively gain confidence in the quality of its requirements, instead of having to cope with too many unwanted behaviours at a time.

Technically, the effectiveness of the STIMULUS simulation engine for generating "really random" traces inside the set of possible traces satisfying the requirements comes from the use of a general constraint solver and of a fair algorithm to pick a random solution inside the solution space.

VIII. *CONCLUSION*

In this paper, we presented STIMULUS, a modeling and simulation tool for the early validation of functional real-time requirements, and we demonstrate its effectiveness for debugging a few requirements typical of the industrial practice, which contain subtle ambiguities and omissions.

STIMULUS is based on modern programming languages and simulation techniques, with many features dedicated to requirement readability. It exploits mature and well-proven research results to make things simple for the users.

It enables engineers to formalize requirements using predefined or user-defined sentence templates, to model environment assumptions and to observe execution traces that satisfy the requirements using the innovative simulation architecture of Fig. 4. It helps finding ambiguous, incorrect, incomplete, or conflicting requirements, as shown on the real-time requirements of an automatic head lights controller of a car provided to us by a software and tools vendor.

Relying on this case study, we discussed the additional benefits that the validation-by-simulation approach for requirements engineering proposed by STIMULUS can bring to the current validation-by-review and validation-by-proof approaches. One can actually observe than in the different domain of control system design, one of the most popular tool is TheMathworks Simulink, which also implements a validation-by-simulation approach.

We did not detail in this paper how to reuse STIMULUS scenario and requirements models for testing black-box real-time systems as depicted on Fig. 3, but this is a hot topic for our customers. Other topics are the automation of some common editing tasks, such as providing functional coverage criteria for requirements, and to automate debugging and testing tasks, such as guiding executions to favor the functional coverage of requirements according to these criteria.

REFERENCES

[1] Jean-Raymond Abrial: Modeling in Event-B - System and Software Engineering. Cambridge University Press 2010, ISBN 978-0-521-89556-9.

[2] Bertrand Jeannet, Fabien Gaucher: Debugging real-time systems requirements: simulate the "what" before the "how". EmbeddedWorld 2015.

[3] Grégoire Hamon, Marc Pouzet: Modular resetting of synchronous data-flow programs. Principles and Practice of Declarative Programming (PPDP'2000), ACM, 2000.

[4] Jean-Louis Colaço, Marc Pouzet: Type-based initialization analysis of a synchronous data-flow language. Journal on Software Tools for Technology Transfer (STTT), 6(3), 2004.

[5] Jean-Louis Colaço, Bruno Pagano, Marc Pouzet: A conservative extension of synchronous data-flow with state machines. Embedded Software (EMSOFT'2005), ACM, 2005.

[6] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, Marc Pouzet. Clock-directed modular code generation for synchronous data-flow programs. Languages, Compilers , and Tools for Embedded Systems (LCTES'2008), ACM, 2008.

[7] Erwan Jahier, Pascal Raymond, Philippe Baufreton: Case studies with Lurette V2. STTT 8(6), 2006.

[8] Pascal Raymond, Yvan Roux, Erwan Jahier: Lutin: A Language for Specifying and Executing Reactive Scenarios. EURASIP J. of Embbeded System, 2008.

[9] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond: Engineering functional requirements of reactive systems using synchronous languages, Symposium on Industrial Embedded Systems (SIES), IEEE, 2013.

[10] Erwan Jahier, Simplice Djoko Djoko, Chaouki Maiza, Eric Lafont : Environment model-based testing of control systems : cases studies. Tols and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 8413, Springer, 2014.

[11] The Mercury programming language. https://mercurylang.org/

[12] Zoltan Somogyi: A system of precise modes for logic programs. Int. Conf. on Logic Programming (ICLP'87), MIT press, 19878

[13] The OCaml programming language. caml.inria.fr

[14] The Haskell programming language. https://www.haskell.org

[15] Jean Goubault. Inférence d'unités physiques en ML. *Journées Francophones des Langages Applicatifs*, pages 3--20. INRIA, 1994.

[16] Andrew Kennedy: Dimension Types. European Symposium on Programming (ESOP'94), LNCS 788, Springer, 1994.