Debugging Real-Time Systems Requirements: Simulate The “What” Before The “How”

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 process iterations as specifications need to be redefined, code rewritten and then redubugged. The major reason for this situation is that no practical tool exists for debugging requirements, and the many tools that exist for requirement management and traceability do not solve this problem. Argosim Stimulus provides an innovative solution for the early debugging and validation of functional real-time requirements. Stimulus relies on a high-level, constraint-based, real-time language to express requirements in natural language and a simulation engine based on a constraint solver to generate and analyze execution traces that satisfy requirements. By visualizing “what” systems will do enables system architects to discover incorrect, ambiguous, missing or incomplete requirements before the design phase starts defining the “how”.

Keywords—Requirement Engineering, Real-time Embedded Systems, Domain Specific Languages, Debugging, Simulation, Constraint Solving

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 practice, requirement specifications 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.

While requirement engineering tools, such as IBM DOORS, exist, they mainly focus on requirement management and traceability, rather than validation. Specification and simulation tools, like UML/SysML or Mathworks Simulink, aim at describing system design and architecture rather than high-level requirements. And, while proof systems, like the Rodin platform based on Event-B, provide expressive languages as well as exhaustive validation features, they are typically hard to use when it comes to debug requirements.

In this paper, we will focus on real-time functional 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.…..

We present Stimulus, an innovative solution for validating functional real-time requirements which provides the following features:

(I) Expresses real-time requirements in a formal yet natural specification language;

(II) Describes environment assumptions using the same specification language;

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

The ability to formalize requirements in a natural language is a necessary condition for users to adopt the tool, while the ability to simulate “what systems shall do” makes requirements validation possible even while writing specifications. This limits specification errors and ultimately reduces costs in the design phase.
As we examine this topic, we will look first at a number of widely accepted requirements quality criteria and examine how to best address them. We will use a small example to illustrate the specification workflow in Section III. Section IV defines the Stimulus requirement modelling language and Section V describes the Stimulus simulation mechanism. We will conclude with perspectives.

II. QUALITY CRITERIA FOR GOOD REQUIREMENTS

Industrial safety standards (ISO 26262, DO-178, EN 50128, IEC 61513, etc.) define a set of quality criteria for writing “good” requirements that follow the IEEE 830 Recommended Practice for Software Requirements Specifications. The subset of functional criteria states that requirements should be:

- Correct (effectively describe the system behaviour)
- Unambiguous (not subject to multiple interpretations)
- Complete (no missing requirements)
- Consistent (no requirement conflicts)
- Verifiable (a test scenario can be written)
- Modifiable (express requirements separately)

Stimulus aims at providing support for checking all these criteria. It is based on a formal specification language to guarantee that requirements are unambiguous and that engineers share clear, common semantics of operations. This language also supports the separate modification of requirements through parallel composition. And, by generating and analyzing possible simulation traces that satisfy requirements, users can check that requirements are both correct and complete, while the solver engine checks consistency by detecting requirements conflicts automatically. Finally, modeling environment assumptions provides a way to verify, i.e. test, requirements against realistic inputs.

III. A RUNNING EXAMPLE

We illustrate in this section the use of Stimulus on a requirement extracted from the DO-253C standard for a Ground-Based Augmentation System (GBAS) involved in aircraft approach operations. The original requirement is described as follows:

When the active Approach Service Type is GAST C, the lateral and vertical deviation outputs shall be invalidated within 0.4 seconds from the onset of any of the following conditions:

a) The Lateral Protection Level (Section 2.3.11.5.2.1.4) or the Lateral Ephemeris Error Position Bound (Section 2.3.11.5.2.1.5) exceeds the lateral alert limit (Section 2.3.11.5.2.1.3.1);

b) The elapsed time from the receipt of any Type 1 message containing the 100-second differential corrections for the ranging sources used in the position solution (Section 2.3.8.1.2) is equal to or greater than 3.5 seconds and the aircraft is within the PAR;

c) The difference between the current time and the reference time of the corrections (derived from the modified z-count) is equal to or greater than 6 seconds and the aircraft is within the PAR.

A. Editing Formal Requirements

Stimulus provides libraries of predefined sentence templates that enable users to edit requirements in a formal yet natural language. Fig. 1 shows the formalization of the above requirement in Stimulus. While conditions a), b) and c) only use logical and numerical operators, the requirement header combines the three following sentence templates:

- “when <expression>, <statement>”
- “from <expression>, <statement>”
- “ensure <expression> within <time>”

We’ll see in the next section that such templates are defined using the Stimulus programming language. Therefore, standard libraries can be extended with new sentence templates that stick to domain-specific needs and practices. The resulting requirement, being very close to the original, is easy to edit, read and maintain.
B. Modeling Environment Assumptions

Even if requirements can be simulated alone, adding signal assumptions often improves the readability of simulation results. Such environment assumptions are modeled with sentence templates, see Fig. 2, which capture common modeling patterns:

- An assumption may be a simple domain range, e.g. for LateralProtectionLevel signal;
- It can be a stability assumption for some average duration, e.g. of AircraftWithinThePAR, where the duration follows a Gaussian normal law;
- It can be an assumption on its variation. For CurrentTime, we use a specific user-defined template “jittering clock” which constrains the derivative to around 1 with some perturbation.

C. Simulate and Debug Requirements

When simulating the requirements together with environment assumptions, Stimulus interactively generates traces that satisfy the corresponding real-time constraints. By observing such simulation results, one can figure out whether the requirements behave as expected, i.e. are correct with respect to IEEE terminology. If not, further analysis of the generated traces may reveal that some requirement is ambiguous or incomplete. If this is the case, the user can complete the requirements and observe new simulations. In our example, a typical issue is to know whether the safety action (invalidate deviation outputs) should be maintained or not, and if so, until when. Fig. 3 shows a possible execution scenario generated by the solver.

D. Test the System Implementation

Once the embedded system has been developed, it can be imported in Stimulus as a black-box system under test. Formal requirements can be turned into test oracles, and environment assumptions can be used to generate numerous test vectors automatically, thereby improving the functional coverage of test campaigns.

IV. SPECIFICATION LANGUAGE

A. Our Approach

To address IEEE 830 principles, the Stimulus language should be as expressive as existing requirements specification languages and as executable as a simulation or a programming language. We chose not to extend an existing specification language like Event-B [1] in order to make it executable, since these languages have their background in logics and are strongly oriented towards proof systems. As a result, they neglect the issues of computationally representing and manipulating the concepts they are based on. We therefore adopted an alternative approach which extends an existing programming language in order to make it expressive enough to define requirements. By doing this, we gain the benefits of using modern programming languages concepts and simulation techniques.

Stimulus combines the concepts of the synchronous language LucidSynchronce [2] and of the language Lutin [3] which are dedicated to the modeling of real-time system environments and test scenarios. LucidSynchronce (like its industrial version SCADE) provides proven and mature concepts for modeling real-time systems, such as dataflow equations, hierarchical state machines, and synchronous composition. Lutin provides the concepts needed for modeling real-time non-deterministic behaviors. We describe in this section the main features of the resulting language.
B. Data: Synchronous Dataflow Equations and Constraints

In LucidSynchrone, the behavior of a variable is described with a dataflow equation. For instance, the dataflow equation
\[
\text{count} := (0 -> \text{last count}) + (\text{if evt then 1 else 0})
\]
specifies (by using :=) that the variable \text{count} is computing the number of occurrences of the Boolean variable \text{evt}. \text{count} and \text{evt} are actually flows: one can refer to their previous value in time with the operator last and -> is the initialization operator. \text{e1} -> \text{e2} evaluates to \text{e1} on reset, to \text{e2} otherwise. Here, the value of \text{evt} is provided by the environment or by a dataflow equation.

In Stimulus, we added the concept of constraints as present in Lutin. For instance, the dataflow constraints
\[
\text{count} = ((0 -> \text{last count}) + (\text{if evt then 1 else 0})); \text{count} <= 10;
\]
specify (by using = and <=) that the values of the flows \text{count} and \text{evt} should be such that \text{count} is less than or equal to 10. In other words, \text{evt} might be true at most ten and may take any value that satisfies this constraint. The values of \text{count} and \text{evt} are not computed but chosen by the solver engine.

Thus Stimulus can model not only real-time computations on known variables but also temporal properties that should be satisfied by unknown variables.

C. Control: Hierarchical State Machines

Like LucidSynchrone, Stimulus provides state machines that are typically used to model running modes or to specify temporal sentence templates. For instance, the automaton of Fig. 4 starts and maintains the previous constraint as long as \text{enable} is true.

Like Lutin, Stimulus also provides non-deterministic transitions. For instance the automaton of Fig. 5 specifies that the variable \text{T} is either increasing or decreasing: in this figure, the numbers 1 and 10 are weights which specify that a switch between the two modes occurs with probability of 1/11. The constraints of the automaton are combined with the interval constraint on \text{T}, illustrating the fact that constraints can be composed modularly in Stimulus. This modularity has non-trivial implication: here, \text{T}=20 at some step would prevent the automaton to enter the increasing state \text{Inc} at the next step, because the conjunction \text{T}>\text{last T} and \text{T}<=20 could not be satisfiable. The backtrack mechanism mentioned in section V addresses this issue.

D. Modularity: Systems, Macros and Sentence Templates

As other synchronous languages, Stimulus encapsulates statements into systems that can then be instantiated at several places. Systems are similar to functions in imperative languages. They are defined by a set of ports (instead of formal parameters) and a set of statements defining or constraining the values of ports. One can for instance define a system \text{Count} that counts the number of occurrences of an event and that can be reused to specify more complex constraints, like
\[
\text{condition} => (\text{Count(evt1)} >= \text{Count(evt2)})
\]
In addition, it is also necessary to have macros, which can be seen as functions that accept statements as parameters. Stimulus provides a robust system of macros with clear scoping and typing rules.

By doing this, common sentence templates, such as the “when” template already used in Fig. 4, can be implemented with parameterized state machines:
Last but not least, formal requirements need to look like textual requirements and not programs, as discussed in section III. 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 the format when <condition>, %<BODY% and to the system Count the format number of occurrences of <event >, the specification of Fig. 4 appears as the sentence

| When enable , number of occurrences of evt | <= 10 |

which is arguably easier to read than Fig. 4. Such a sentence template can be graphically edited and modified with Drag&Drop, Cut and Paste operations.

Stimulus includes a powerful library implementing the ubiquitous concepts and temporal patterns encountered in real-time requirements, equipped with such formats fitting the standard ways of writing requirements. This library supports the claim (I) of “a formal yet natural specification language” and can be extended by users in order to address to internal corporate guidelines.

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 Simulink. This enables developers to visually describe the architecture of a system graphically with the flow of information in it. For instance, let’s look at the test bench of a simplistic heater controller:

![Test bench confronting a system with its requirements](image)

F. Scalability: Controlling Constraint Propagation

Stimulus can be viewed as a constraint-programming language like Prolog dedicated to real-time systems. Prolog programming experience showed that an overly declarative style of programming leads to programs that were difficult to understand and debug. Therefore, successors of Prolog, like Mercury [4], provided concepts to better control the propagation of constraints and the order of computations [5]. Stimulus follows this philosophy: constraints should be kept local in specifications and solved incrementally rather than globally. This often reflects the way of thinking of the user, thus preserving scalability for debugging; it also keeps the size of the constraint systems under control, thus preserving scalability in terms of efficiency.

For this purpose, Stimulus enables the user to encapsulate a set of statements in a solving box, parameterized by a set of variables to be produced:

- these variables are considered as unknown inside the box and known outside the box: the solver will assign them values satisfying the constraints in the box;
- the other variables involved in the box are required to be known inside the box.

For instance, one can write

```plaintext
solve x, y in
  x+y>=0; x+y<=3; x-y>=0; x-y<=4
end;

z := exp(x+y*y); // x and y are known here
w>=z and w<=2*z;
```

These statements specify that it first picks values for x and y that lie inside a rotated rectangle, then computes the exponential of a non-linear expression on them and assigns it to z, and at last constrains w with respect to z. One can thus perform in sequence solving and much cheaper computation phases.

G. Static Checks: Typing and Similar Features

Stimulus uses the principles of the type inference approach of typed functional languages like OCaml [6] and Haskell [7] to minimize annotations in specifications and still provide powerful static checks.

Types are inferred and include usual scalar types (Boolean, enumerated and numerical types), arrays and records. Numerical types may be specialized with a physical dimension: Stimulus implements the approach of [8] for checking the consistency of expressions w.r.t. physical dimensions and units. For instance, if pos is of type [length] real and dt is of type [time] real, the expression pos+dt is of type [length/time] real.

As other synchronous compilers, Stimulus checks the absence of causality cycles; this makes sense only for known variables that are produced by assignments or solving boxes, or in other words by computations.

- For instance, \(x := 2*y; y := x + 1;\) induces a causality cycle between the known variables \(x\) and \(y\): the problem is that there is no way to order the two computations.
- but \(x := 2*y; y := x + 1;\) is seen as the conjunction of two constraints that do not raise a causality issue and that has the solution \(x = -2, y = 1\).

Related to this analysis is the inference of the directions of the ports of a system:

- a port is either an input—its value is needed but not produced inside the system, hence it should be produced outside the system,
- an output—it is produced by the system or
- acasual—the system just adds new constraints involving it.
Again, such an inference mechanism reduces the burden of providing annotations on the user.

V. COMPILATION PROCESS AND SIMULATION ENGINE

Stimulus specifications are first compiled to a lower-level bytecode, and then simulated.

The compilation process follows the principles of [2] 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 compiler also takes care of allocating memory for variables. The resulting bytecode has roughly the abstract syntax:

```
bc := switch(ck)(bc, bc, ...)  Non-deterministic choice indexed by clock
| bc ; bc                     Sequence
| Constraint                  Atomic constraint
| Assign | ExternalCall         Atomic computation
```

The simulation engine combines

- an exploration algorithm for resolving the non-deterministic choices induced by state machines;
- a constraint solver for resolving the non-determinism induced by constraints on variables.

The exploration algorithm follows the principles of the Lurette simulation engine [11] for trying the branch of a choice, executing its statements and backtracking in case of unsatisfiable constraints to try another branch. This backtrack mechanism is needed for instance for simulating the nondeterministic state-machine of Fig. 5, as explained in Section IV.C. Backtrack occurs only inside a reaction step: Stimulus never “rewinds” the time.

The constraint solver accepts any propositional formula on Boolean, enumerated and numerical variables, provided that numerical expressions are linear in the unknown variables. It exploits BDDs [12] and convex polyhedra to perform the satisfiability checks requested by the exploration algorithm, and to pick solutions for variables with a uniform random policy. An SMT (SAT Modulo Theory) solver could be used as well, provided that it is able to pick solutions with a policy that is “random enough”. Ideally one wants indeed to explore all the possible executions that are compatible with the constraints.

In practice, as there is an infinite number of such executions, the simulation engine should be at least able to generate any of them, without excluding some of them forever.

VI. CONCLUSION

In this paper, we have presented Stimulus, a modeling and simulation tool for the early validation of functional real-time requirements. Stimulus enables engineers to formalize requirements using predefined sentence templates, and to observe execution traces that satisfy the requirements. It helps finding ambiguous, incorrect, incomplete, or conflicting requirements, providing support for the verification of IEEE 830 requirements quality criteria. We have illustrated the workflow for editing, debugging and validating an effective real-time requirement extracted from the aeronautic DO-253C standard. We have then described the underlying technical foundations of Stimulus.

From a technical point of view, we believe that the key originality of our approach is to closely combine the scientific concepts and tools of two distinct research communities, namely the synchronous language community with the LucidSynchron [2] and Lutin [3] languages, and the formal verification community with the use of BDDs and convex polyhedra.

From a user point of view, we paid attention to make the Stimulus language as simple as possible by adding many features dedicated to requirements readability. In particular, sentence templates described in Section IV.D provide an effective way to extend the standard library with user-defined templates, or to customize predefined templates.

Stimulus has been successfully evaluated in several case-studies from the automotive, railway, avionics and nuclear industries. We plan to automate some common debugging tasks, such as isolating conflicting requirement or guiding executions towards requirement violation or satisfaction. Finally, we plan to improve the functional coverage of test vectors generated from environment assumptions and test scenarios (by testing limit values, covering all states, etc).

REFERENCES