Attitude Control of the Space Shuttle: A Retrospective Example on Model-Based Design and Verification Processes *

Raphael Cohen[1−2], Hamza Bourbouh[3], Guillaume Brat[3], Eric Feron[1], and Pierre-Loic Garoche[2−3]

[1] Georgia Institute of Technology, GA 30322, USA
[2] Onera – The French Aerospace Lab, Toulouse, France
[3] NASA Ames Research Center

Abstract. In the past years, Software has been one of the areas of engineering with the biggest impact on systems performance. For the aeronautical and aerospace industries, an important part of both the cost and the innovations are software related. Modern tools, programming languages and development techniques have emerged, letting the possibility to design bigger and more complex software systems faster. Thus, Software are growing in complexity and its development cycle is evolving through time. In this article, we revisit a famous system, the attitude control of the NASA Space Shuttle as described in a 1982 document by Draper Laboratory. At that time model based design, code generation and formal methods were not developed as they are now. Here we propose the redesign that system using modern tools and showing how model-based design and formal verification can be applied on realistic system. This paper presents the structure of the control and its implementation as a Simulink model fitted with formalized specifications. We also present analysis result of the Simulink-based analyzer CoCoSim.

Keywords: Simulink · Software Verification · SMT Solvers · Model-Based Design · Code generation · Control System · CocoSim

1 Revisiting the Space Shuttle Attitude and Orbital Control System with Model-based Design and Formal Verification

We discovered a gem: a 35 years old unclassified report from Draper Lab detailing the Attitude and Orbital Control System (AOCS) of the famous Space Shuttle. We propose here to revisit its design following a modern model-based development relying on auto-coding and early uses of formal verification.

This article gives a brief overview of the model and the verification activities performed. A more complete access to the report, the produced model and the identified and formalized requirements can be found at `https://cavale.enseeiht.fr/spaceshuttle`.

We relied on Matlab Simulink to perform the initial design and then refine the model to fit with our CoCoSim toolbox supporting code generation and formal verification. In contrary to other approaches that search to prove that the compilation preserves faithfully the original semantics [2, 9], we rather believe that formal methods will find their ways in the industry practice when it will complement existing development process, which are, nowadays, largely based on code generation from dataflow models, ie. the DO-178 qualified compiler KCG for Scade or the Embedded Coder for Matlab Simulink.

Our proposal relies on the formalization of requirements as model components, a natural language for the control system practitioner, and the use of model-checking to check early the validity of these requirements, without the need to wait for the final code to be produced. Such requirements ought to be compiled throughout the development process and revalidated at any stage.

Researchers in formal verification always look for realistic examples of control systems since industry hardly shares its own models. As an example the NASA TCM [3] was design to share such a

* The work was partially supported by project ANR-17-CE25-0018.

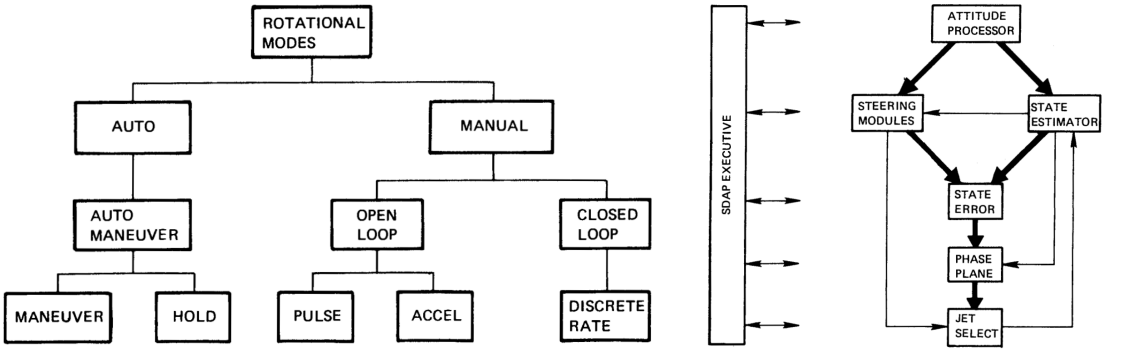
model. Unfortunately, while mimicking a realistic development process and sharing with the community a complex example, it lacked the full documentation typically associated to such a development. In industry these documentation, the specification, drive all the development.

In this specific case we are given with this initial specification of the Space Shuttle AOCS and we use it to illustrate the advances of model-based design and formal verification. We hope that the community will gather around it and use it to compare approaches or relative efficiencies of technical proposals.

The paper is structured as follows. The next section presents key elements of the Space Shuttle AOCS. Then, in Section 3, we illustrate on a single requirement our formalization, at model level, of the specification document. Then Section 4 addresses the required transformations needed by our toolbox as well as early results on the formalized requirements while Section 5 concludes.

2 Space Shuttle Attitude Control as a Simulink Model

Simplified Digital Autopilot (SDAP) Executive. Fig. 1a presents the different modes in which the Simplified Digital Autopilot (SDAP) of the Space Shuttle can operate. Each of them is described in the specification document and is modeled as a Simulink component. The most complex one being the "AUTO MANEUVER" sub-mode within the "AUTO" mode. Its architecture is depicted in Fig. 1b:



(a) Moding possibilities (b) Auto-Maneuver Modules architecture

Fig. 1: SDAP Architecture

The SDAP executive will command and execute the software logic at a frequency of 12.5 Hz. After collecting the current state from the IMUs (current attitude), the DAP, depending on the switches values entered by the crew, will send appropriate commands to the actuators (jets) to bring about a desired state. The crew can choose the behavior they desire using set via keyboard entries, and by push-buttons moding discrete, such as the choices between primary/vernier jets and automatic/manual attitude control mode. The Simulink model modeling the whole embedded architecture of the attitude control software is presented in Fig. 2. In this Simulink model, a block is associated to each modules.

Auto Maneuver Module. The Auto Maneuver module will allow the crew to perform attitude change maneuvers or attitude hold depending on their need. The crew would enter to the computers a commanded inertial attitude and based on this desired attitude and the current one, the SDAP will execute the appropriate maneuver. In order to execute the maneuver, this module, from the switches, crew inputs, will compute the desired and current attitude and rate, using quaternions. Once those values computed, they are sent to the phase plane module.

Phase Plane and Jet Selection Modules. The phase plane module represents the heart of the attitude control feedback. For this, given the errors in attitude and rate, θ_e and ω_e for each axis, an index region is computed (as shown in Fig. 3). Then, based on this region and the controller state at the previous time a variable called "ROTATION COMMAND" is computed. The jet selection module, from the "rotation command" value computed in the phase plane module, will compute 17 booleans corresponding to the command sent to the primary jets (11 jets) and vernier jets (6 jets). Each selected boolean variable will result in firing the associated jet for a time duration of 0.08 sec.

Rotational Dynamics – Plant. In order to model the closed-loop behavior of the Space Shuttle and to do simulations we constructed a dynamical system of the plant, following the model presented in the specification document appendices. For this, we used a simplified linear system. Let ω the rotation speed of t he space shuttle. we have the dynamics:

$$\tau = I \cdot \dot{\omega} \tag{1}$$

With: τ the total moment acting on the vehicle, I the vehicle inertial matrix. Thus, after discretization, we end up with the below relation, which will be the one implemented:

$$\dot{\omega} = I^{-1} \cdot \tau \tag{2}$$

$$\Delta\omega = I^{-1}\tau \cdot \Delta t = I^{-1}\tau \cdot 0.08 \tag{3}$$

3 Expressing Requirements as Semantics Blocks

A key element to support formal verification of safety-critical software is the formal expression of the specification. This step can be difficult in general when targeting the verification of imperative code, for example the expression of the specification as ACSL [1] function contract for C code. However, in the specific case of synchronous dataflow models, such as Matlab Simulink, Scade or Lustre, the specification can be easily expressed in the model language as synchronous observers [7, 10]. Furthermore Assume-Guarantee contracts, ie. Hoare triples [8], can be lifted to dataflow semantics. CoCoSpec contracts [4] provides such as extension for Lustre and can be expressed at Simulink level. A regular Simulink subsystem can be used to encode such contracts, building boolean signals to encode Assume/Guarantee predicates.

We identified and labeled in the Space Shuttle report numerous sentences that could act as requirements. Each of these requirements is then expressed as a Lustre CoCoSpec contract and can then be expressed in the Simulink model.

Let us illustrate the approach on one specific requirement of the space shuttle. As mentioned above two sets of thrusters are used to control the attitude of the Space Shuttle. The primary jets being way more powerful than the vernier. We identified in the Jet selection module, at Section 3.8.2, the *Req.s*3.8.2.*p*63.1 stating that "The two types of thrusters may not be used simultaneously".

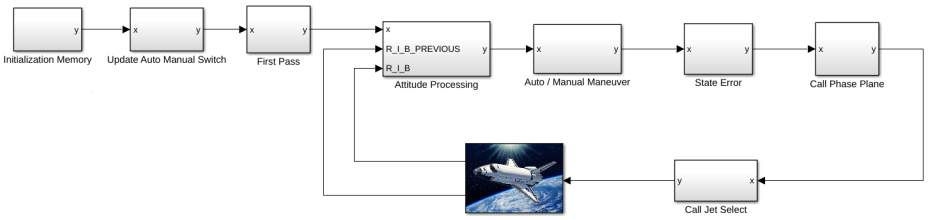


Fig. 2: Main Simulink Model of the Embedded Software

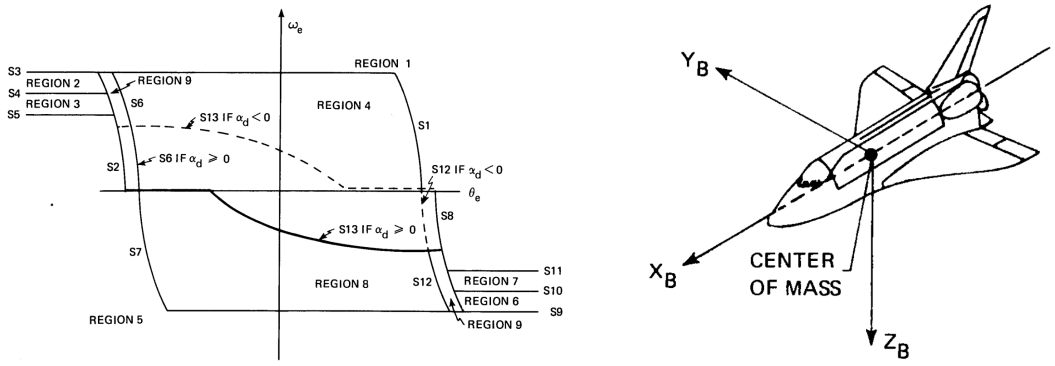


Fig. 3: (a) Phase Plane switch lines and regions. (b) Body Frame of the Shuttle.

From this, we chose to represent this condition by the three properties below. The variables "Vernier", "Primary", arrays of boolean of appropriate sizes, represent the state of the vernier and primary jets.

$$\Big(\text{primary}(1) \;\; \text{OR} \;\; \ldots \;\; \text{OR} \;\; \text{primary}(11)\Big) \;\text{NAND}\; \Big(\text{Vernier}(1) \;\; \text{OR} \;\; \ldots \;\; \text{OR} \;\; \text{Vernier}(6)\Big) \tag{4}$$

$$\Big(\text{primary}(1) \;\; \text{OR} \;\; \ldots \;\; \text{OR} \;\; \text{primary}(11)\Big) \implies \mathit{vernierSW} = \mathit{ON} \tag{5}$$

$$\Big(\text{Vernier}(1) \;\; \text{OR} \;\; \ldots \;\; \text{OR} \;\; \text{Vernier}(6)\Big) \implies \mathit{vernierSW} = \mathit{OFF} \tag{6}$$

The property 4 encodes the fact that only one kind of jet can be used at a time. Couples of jets of the same type could be used simultaneously. Property 5 enforces that if one or more primary jet are fired, the vernier switch has to be on. Finally, the property 6 is translating the same property but for the vernier jets: if one or more vernier jets are fired, the vernier switch has to be off. These logical statement are first expressed as CoCoSpec contract in Fig. 4 then as Simulink observers in Fig. 5.

```
(* Req Id: req_s3_8_2_p63_1
   Text: The two types of thrusters may not be used simultaneously
   Origin: Section 3.1 SDAP Executive, p63
   Simulink Component Name: CallJetSelect.slx
   Requirement PATH: "CallJetSelect/ PRIMARY VERNIER Req" *)
contract req_s3_8_2_p63_1 (PRIMARY_VERNIER_SW: bool;) returns (PRIMARY_JET_COMMAND:bool^11;
                                                               VERNIER_JET_COMMAND:bool^6;);
let
      var is_primary : bool = (PRIMARY_JET_COMMAND[0] or ... or PRIMARY_JET_COMMAND[10]);
      var is_vernier : bool = (VERNIER_JET_COMMAND[0] or ... or VERNIER_JET_COMMAND[5]);
      guarantee is_primary => PRIMARY_VERNIER_SW;
      guarantee is_vernier => not PRIMARY_VERNIER_SW;
      guarantee is_primary NAND is_vernier;
tel
```

Fig. 4: Formalization of Req. *Req.s*3.8.2.*p*63.1 as a CoCoSpec contract.

4 Verifying specification using CoCoSim

CoCoSim is an automated analysis and code generation framework for Simulink and Stateflow models. It is based on a compiler architecture and provides means to compile the input Simulink model into the synchronous dataflow language Lustre, used as an intermediate formal language. Once both the model and its annotations blocks are expressed as Lustre and CoCoSpec contracts, respectively, one can rely on Lustre analysis tools such as the model-checkers Kind2 [5] or Zustre [6]. This enables the validation of the specification at model level.

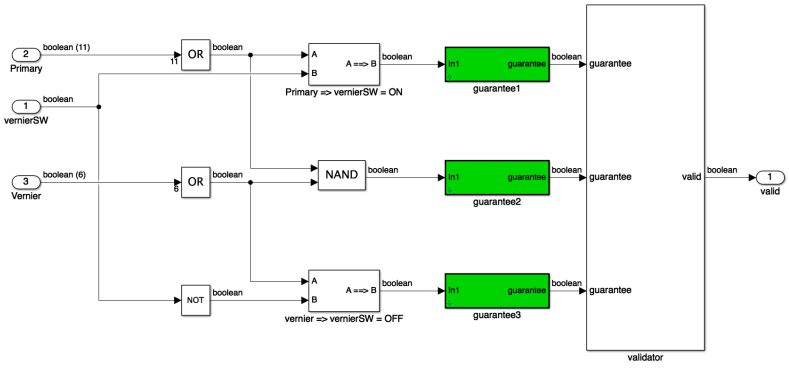


Fig. 5: Simulink Observer implementing Req. *Req.s*3.8.2.*p*63.1, preventing simultaneous use of jets

4.1 Rewrite the model as CoCoSim compliant

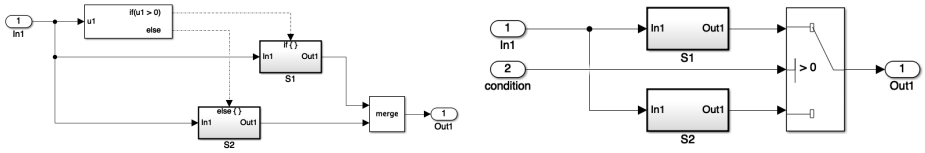
The current implementation of CoCoSim supports 100 Simulink blocks including about 20 blocks that are simplified in the pre-processing step into basic blocks. In addition, inside our Simulink model we used some of libraries from the Aerospace toolbox performing, for example, quaternion arithmetics (such as Quaternion Multiplication, Quaternion Conjugate, etc). CoCoSim supports these blocks as these are masked-subsystems and the content of the subsystem is based on basic supported blocks.

Unsupported blocks. A first version of the model also relied on *unsupported blocks* such as Matlab functions or Data Store Memory, Data Store Write, and Data Store Read. The former were used to describe sequential algorithms, with bounded loops and nested If-else statements. We rewrote all Matlab Function code into pure Simulink blocks. The latter where used as global variables: in general Simulink model provides a hierarchical graphical view of the model but this structure is typically not enforced. Our choice of using Lustre as an intermediate language imposes to maintain the hierarchy of the model but also support modular analysis and therefore could address issues of scalability of the formal verification. We transformed the signature of each component to propagate these signals along modules, removing their global definitions.

Verification efficiency. In addition to making the model compatible with the toolchain, good modeling practices could greatly support the CoCoSim analyses, such as setting the correct DataType machines on the signals (e.g. boolean, double single, int8, etc). Moreover some blocks could be soundly simplified. For instance, If-Else blocks (see Fig. 6) can be substituted by Switch blocks when the Action Subsystems linked to If-Else have no memory blocks (e.g Unit Delay) inside. In fact, Action Subsystems are conditionally executed subsystems. It means that their execution is controlled by a signal (in this case, their associated If-else condition). In Lustre, a conditionally executed subsystem can be translated as the Lustre automaton presented below but the equivalent yet simpler expression using Switch block (Fig. 6) will be translated in Lustre as: `out = if condition then S1(in)else S2(in);` The initial Lustre automaton would produce more variables, more clocks and local memories, making the analysis more difficult.

```
automaton s1
  state S1_IS_Active:
  unless (not condition) restart S1_IS_INACTIVE
  let
    --call S1 subsystem;
  tel
```

```
state S1_IS_INACTIVE:
unless condition resume S1_IS_Active
let
 --reset or resume outputs previous values
tel
```



(a) If-else Block In Simulink (b) Equivalent version using Switch block

Fig. 6: Various encoding efficiency

Req. ID	Simulink Component	Text
Req_p63_1	Call Jet Select	The two types of thrusters may not be used simultaneously
Req_p19_1	Auto Manual Switch	If the hand controller is deflected in any axis, the SDAP automatically switches to manual mode
Req_p19_5	Auto Manual Maneuver	When the maneuver mode is changed from manual to auto, if the bypass flag is ON, it is set to OFF and the auto maneuver initialization flag is set to ON.
Req_p27_1	Auto Manual Maneuver	Auto Maneuver tests the rotation angle rotation_angle_delta_theta against two numerical criteria. If rotation_angle_delta_theta is larger than $y = SCALARBIAS + 2 * Deadband$, the module places itself in the maneuver mode; if rotation_angle_delta_theta is less than $x = SCALARBIAS + Deadband$, the hold mode results.

Req. ID	Simulink Component	#blocks	SLDV Result	SLDV Time	CoCoSim Result	Total Time	Lustre Gen. Time	Verification Time
Req_p63_1	Call Jet Select	34	Valid	27s	Valid	21s	19s	2s
Req_p19_1	Auto Manual Switch	8	Valid	7s	Valid	12s	10s	2s
Req_p19_5 Req_p27_1	Auto Manual Maneuver	589	Valid	23s	Valid	34s	30	4s

Note that total time reported for CoCoSim is detailed as compilation time to Lustre and actual verification. SLDV is the model-checking tool provided by MathWorks.

Table 1: Selection of requirements and verification results

4.2 V&V at model level, based on formalized requirements as Cocospec contracts.

Selected and formalized requirements were provided to the solvers and their property evaluated. While this process can be troublesome, it is largely automated here, injecting the CoCoSpec requirements in the model and applying model-checkers to the result Lustre model. Traceability maps allow to interpret back the result at model level, in Simulink. Table 1 provides a very short overview of formalized requirements and their proved validity.

The initial set of 49 requirements can be found at `https://cavale.enseeiht.fr/spaceshuttle`.

5 Conclusion

We presented an interesting use case: a closed loop description of the the attitude and orbital control system of the Space Shuttle. All documentation, identified requirements and Simulink models are available online. The work is only preliminary and the model could be still used for various activities: computing attitude trajectories, performing control-level closed-loop analyses, or even floating point accuracy verification.

References

[1] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language*. frama-c.cea.fr/acsl.html. 2008. URL: frama-c.cea.fr/acsl.html.

[2] Timothy Bourke et al. "A formally verified compiler for Lustre". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 2017, pp. 586–601. DOI: 10.1145/3062341.3062358. URL: https://doi.org/10.1145/3062341.3062358.

[3] Guillaume Brat et al. "Verifying the Safety of a Flight-Critical System". In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. 2015, pp. 308–324. DOI: 10.1007/978-3-319-19249-9_20. URL: https://doi.org/10.1007/978-3-319-19249-9%5C_20.

[4] Adrien Champion et al. "CoCoSpec: A Mode-Aware Contract Language for Reactive Systems". In: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*. 2016, pp. 347–366. DOI: 10.1007/978-3-319-41591-8_24. URL: https://doi.org/10.1007/978-3-319-41591-8%5C_24.

[5] Adrien Champion et al. "The Kind 2 Model Checker". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 2016, pp. 510–517. DOI: 10.1007/978-3-319-41540-6_29. URL: https://doi.org/10.1007/978-3-319-41540-6%5C_29.

[6] Arnaud Dieumegard et al. "Compilation of synchronous observers as code contracts". In: *SAC'15*. 2015, pp. 1933–1939.

[7] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. "Synchronous Observers and the Verification of Reactive Systems". In: *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993*. 1993, pp. 83–96.

[8] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[9] Xavier Leroy. "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant". In: *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54. URL: http://xavierleroy.org/publi/compiler-certif.pdf.

[10] John M. Rushby. "The Versatile Synchronous Observer". In: *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*. 2012, p. 1. DOI: 10.1007/978-3-642-33296-8_1. URL: https://doi.org/10.1007/978-3-642-33296-8%5C_1.