# Towards Using Formal Methods in Prototyping: Advantage or Impediment?

Sebastian Schirmer
*Unmanned Aircraft*
*German Aerospace Center (DLR)*
Brunswick, Germany
sebastian.schirmer@dlr.de

Tino Teige
*BTC Embedded Systems AG*
Oldenburg, Germany
tino.teige@btc-es.de

Christoph Torens
*Unmanned Aircraft*
*German Aerospace Center (DLR)*
Brunswick, Germany
christoph.torens@dlr.de

Udo Brockmeyer
*BTC Embedded Systems AG*
Oldenburg, Germany
udo.brockmeyer@btc-es.de

***Abstract*—In aviation and other safety-critical domains, software faults are unacceptable. A means of detecting and avoiding these faults is to use formal methods. Although formal methods strongly contribute to the reliability and robustness of the system, some drawbacks prevent their general usage. A drawback is their reputation to be hard to apply for non-experts. Non-experts have to be familiarized with the tools to efficiently make use of them. But is this reputation still valid? Over the years, formal methods tools have evolved. They are capable to analyze more complex system properties. Further, their user experience was addressed by industrial companies to actually allow non-experts to profit from the advantages of formal methods.**

**This paper represents the first step towards putting the mentioned assumption under test by trying to use formal methods for prototyping. We propose an approach for software prototyping which makes use of the formalization of requirements. We depict advantages and discuss first results of evaluating the commercial tool BTC Embedded*Platform*® that we were able to use without cost in a project cooperation. We plan to continue the project cooperation to answer the headline in future.**

***Index Terms*—prototyping, formal specification, monitoring**

## I. Introduction

In software development, validation is a difficult and error-prone task. There are several different development models that lead through the software life cycle and support the development of the software product. Common to most of these methods is a set of requirements from which code and test cases are created. Complementary, when developing safety-critical software, recommended software standards such as DO-178C [1] also recommend traceability in both directions from requirement to code and from code to test cases. Further, it is known that the earlier in the development process a bug is found the cheaper it can be fixed.

Often a proof of concept is developed with reduced rigor in form of a prototype before the final software product is created. In aviation, ARP4754A [2] even proposes prototyping to identify incorrect or missing requirements. One of the main reasons for developing a prototype is improving the understanding of the target product. Only with this knowledge it is possible to define good software requirements. Otherwise, the values used in requirements are unknown or too fuzzy. To derive these values, we think that a structured way of prototyping is desirable during the prototyping process, i.e. consistent and documented adaptation of threshold values.

At the department Unmanned Aircraft of German Aerospace Center (DLR), prototypes are built to foster autonomy of unmanned aircraft system. This also incorporates to consider AI methods and their validation. One problem in AI is that the function is not written as transparent and human-readable code which can be manually checked and traced. Instead, the function is learned and tested based on data and, therefore, often considered as black-box. Currently, there is a trend towards simulation validation of AI due to the lack of other means. To check black-box systems, e.g. during simulation, one approach is runtime monitoring [3], depicted in Fig. 1. There, the inputs and outputs of the black-box are send to a monitor. The monitor is generated based on a formal specification of desired system properties and evaluates whether the requirements are met by the black-box [4]. Further, in case of an AI component, a monitor feedback loop, as additional qualitative or quantitative features, can also enhance the AI learning phase [5]. For runtime assurance purposes, monitoring plays an essential role. Regulatory agencies are investigating the means to ensure comparable safety of AI components, which are in line with common practices. One promising means seems to be some version of the simplex architecture [6] to sandbox the untrusted AI function. There, a decision module decides whether to switch from an untrusted function to a trusted function. Similar to the fuzzy values mentioned above, the decision module requires accurate threshold values to switch safely and efficiently. An extension to the simplex architecture [5] was recently presented which incorporates neural aspects of AI.

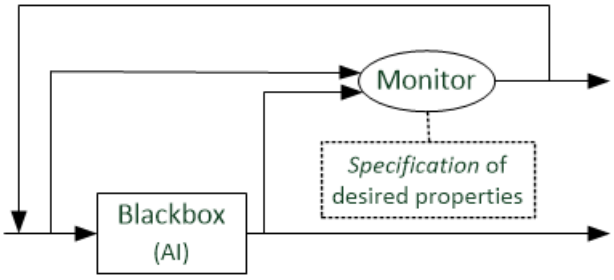


Fig. 1. Using runtime monitoring to check black-box systems.

In summary, an approach which allows to early identify requirement violations and to safely (documented, consistent) adapt requirement threshold values to capture or even bound the behavior of the target (AI) application is highly desirable for both developing an end-product as well as prototyping. In this paper, we propose an approach based on the formalization of requirements and depict its possible advantages. Further, we point out first insights whether the commercial tool BTC Embedded*Platform*® can be used for the proposed approach.

## II. Related Work

Imprecise or incorrect requirements is one of the root causes of software errors and the associated costs to fix a software error in late stages of development is highest for errors introduced during requirements elicitation [7][8]. A requirements-based testing process that was supported by formal methods has been shown to significantly reduce the number of issues before the implementation phase [9][10]. Previous work targeted this by formalizing requirements first in a semi-formal natural language template and finally in a temporal logic [11]. The formal requirements were then model-checked against a manually developed model of the software. Although this approach was extremely helpful to understand the system, it required a lot of manual development effort and the approach did not include support for code and test development. With this work, we continue these efforts and include additional aspects.

## III. Approach

Often during prototyping, assumptions and requirements on the system are implicitly given. Still, we think they should be explicitly stated during the prototyping process. Especially functional requirements are interesting for prototyping: *every 100Hz the controller has to output a value;* control-flow properties like *if the pilot requests ascending of the drone then the drone shall increase height within 0.2 to 0.5 seconds;* or *if an obstacle is detected above then the drone shall stop to increase height within 0.1 seconds;* or even high-level assumptions like *during daytime tracking of an object works*. They can actually help developers to avoid false, e.g. outdated due to system changes, assumptions on the current software state. Of course, writing and checking requirements on the prototype level imply an additional workload. However, it can guide towards the source of error. Also, after prototyping, written requirements offer a good starting point for creating the full set of requirements. In general, prototyping should not turn into an end-product development, therefore a lightweight approach is desirable.

The approach focuses on the formalization of such functional requirements. As a result, the requirements could be depicted as an automaton or formula. This is an additional step which needs to be light and efficient. But the main advantages of formalized requirements are: First, they offer an unambiguous documentation. Second, at an early stage consistency can be automatically checked and, therefore, later code iterations potentially avoided. Third, basic test cases can be automatically generated. Forth, monitors can be generated for each property and attached to the code, checking the adherence to the property online. In the next subsection, we describe one way how the formalization can be achieved. The approach is shown in Fig. 2 and explained in more detail in Subsection III-B.

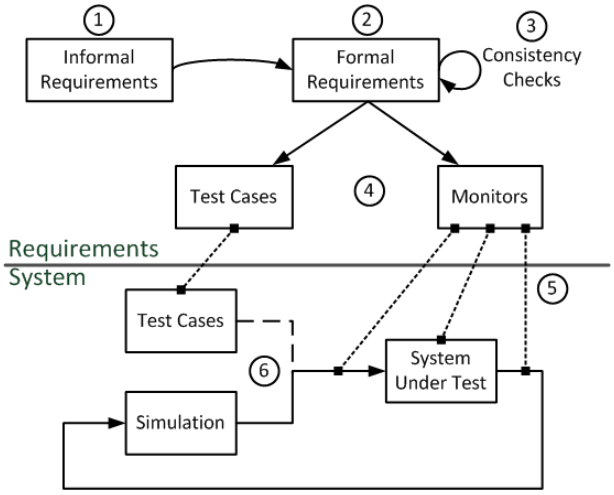


Fig. 2. Proposed approach for prototyping.

### A. Tool Support

BTC Embedded*Platform*® (EP) is a tool suite for specification, testing, and verification of requirements for Simulink® and TargetLink® models and production code, currently mainly used in the automotive domain. EP has been successfully certified by German TÜV Süd as fit for purpose for the usage in safety-critical software development projects according to IEC 61508-3:2010, ISO 26262, EN 50128, IEC 62304 as well as ISO 25119. However, there is no dedicated tool qualification support for the aerospace domain, as of yet.

The workflow for the use case of this paper is shown in Fig. 3. Textual requirements are first imported into EP. Then the BTC specification method guides the user during the complex conversion phases from an informal textual requirement to a uniquely defined and correctly formalized requirement. One particularity of the method is that the underlying formalism called *Simplified Universal Pattern* (SUP) inherently reflects the intrinsic nature of an informal requirement by describing a temporal trigger-action relationship, namely in an intuitive and graphical way.

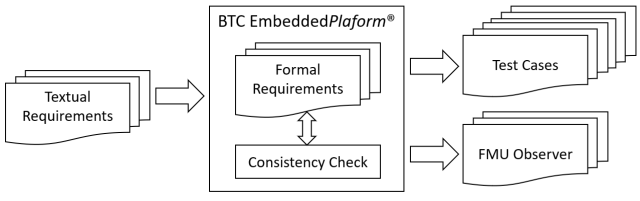


Fig. 3. Workflow using BTC Embedded*Platform*®.

We briefly introduce the idea behind SUP by means of an example. For more details about the syntax and semantics of SUP the reader is referred to [12][13]. Fig. 4 gives two examples of formal requirements using the temporal trigger-action relation of the graphical SUP language. The SUPs FR-1 and FR-2 formalize the textual requirements from Section III "*If the pilot requests ascending of the drone then the drone shall increase height within 0.2 to 0.5 seconds.*" and "*If an obstacle is detected above then the drone shall stop to increase height within 0.1 seconds.*", respectively. An example execution of FR-1 is shown in the upper part of Fig. 5: in step 1 the pilot requests ascending of the drone and thus the trigger of the SUP is activated. This in turn means that the action is expected to occur two to five steps later (where the step size is defined to be 0.1 seconds). In fact, the height is increased in step 5 meaning that the action part of the SUP fires which leads to a fulfillment of the SUP in the same step.

Based on formalized specifications, fully automated formal checks can be executed in order to find and fix problems in the requirements set at a very early design stage, which leads to greater product quality with less cost compared to traditional processes. As indicated by Fig. 3, we consider the *consistency* check of requirements within this paper, i.e. we aim at finding any contradiction within the requirements fully automatically [14]. When applying consistency analysis on the two SUPs of Fig. 4, EP reveals an inconsistency witnessed by a violating execution shown in the lower part of Fig. 5: the pilot requests ascending of the drone in step 0. Due to FR-1, this enforces to increase height at least in step 2 and at most in step 5. In the concrete inconsistency witness, height is not increased in steps 2, 3, and 4, i.e. the only chance to fulfill FR-1 is to increase height in step 5. However, in step 5 an obstacle occurs which triggers FR-2. A valid behavior by FR-2 is to stop to increase height in the same step which in turn contradicts with finishing the action of FR-1. This establishes an inconsistency between FR-1 and FR-2. Though the detection of inconsistencies is fully automatic, their resolution is a manual task but with the support of a concrete counterexample. In the example above, it was most likely forgotten to consider detected obstacles for the pilot request in FR-1. One potential solution would be to define an *exit* (cf. [12][13]) in FR-1 whenever an obstacle is detected, with the effect that observation of FR-1 is restarted (without violating or fulfilling the SUP) whenever an obstacle occurs.

Finding requirement-specific test cases, i.e. test cases that

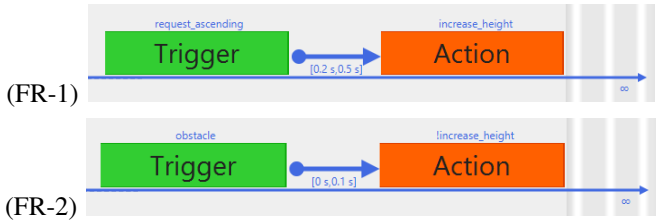


Fig. 4. Two SUPs FR-1 (upper) and FR-2 (lower).

| Name | Mode | Step 0 | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 |
|---|---|---|---|---|---|---|---|
| Formal Requirement | Status | | | | | | Fulfilled |
| Commitment | SUP phase | Startup→▽(Tr) | →Tr→▽(Act) | ▽(Act) | ▽(Act) | ▽(Act) | →Act→✔→▽(Tr) |
| request_ascending | input | 0 | 1 | 0 | 0 | 0 | 0 |
| increase_height | output | 0 | 0 | 0 | 0 | 0 | 1 |
| obstacle | input | 0 | 0 | 0 | 0 | 0 | 0 |

| Name | Mode | Step 0 | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 |
|---|---|---|---|---|---|---|---|
| Formal Requirement | Status | | | | | | Violated |
| Commitment | SUP phase | Startup→▽(Tr)→Tr→▽(Act) | ▽(Act) | ▽(Act) | ▽(Act) | ▽(Act) | → |
| request_ascending | input | 1 | 0 | 0 | 0 | 0 | 0 |
| increase_height | output | 0 | 0 | 0 | 0 | 0 | 0 |
| obstacle | input | 0 | 0 | 0 | 0 | 0 | 1 |

Fig. 5. Example executions of SUPs.

actually test the requirements, most often is a *manual* and thus very time-consuming task. As shown in Fig. 3, EP facilitates the *automatic* generation of test cases by taking into account the intended behavior described by the requirements, leading to high-quality test cases, cf. [14]. For the SUPs FR-1 and FR-2, a generated test case is shown in the upper part of Fig. 5.

Another capability is to export executable monitors, also called *observers*, from formal requirements as depicted in Fig. 3. These observers implement the behavior of the formal requirements and establish a very flexible way of monitoring the correctness of the behavior of the actual system under test in external simulation environments. Technically, these observers are exported as a *functional mock-up unit* (FMU) in the tool independent FMI standard[1], cf. [15].

### B. Prototyping Using Formal Requirements

We propose an iterative prototyping approach (cf. Fig. 2):

1. Informal requirements on the application are collected.
2. We formalize the requirements using EP. Note that EP focuses on functional requirements. Requirements on the system design, e.g. structural redundancies, cannot be expressed in general.
3. Then, formal requirements are checked for consistency. An automatic check is highly desirable since the software is build iteratively and variables can change at any point in the prototype development. Also, especially, wrong timing assumptions are hard to find during later iteration. Imagine the requirement on component c *when input a is set, output b should be set within 50ms*. At a later development phase, component c is divided into consecutive subcomponents c1 and c2 with requirements *if input a is set then a1 is set after 25ms* and *when input a1 is set then b is set within 25ms*, respectively. Due to the automated check of the formal requirements, we can automatically detect an inconsistency by finding a trace falsifying the initial requirement on c. The required time for the second property is unbounded, *after 25ms* and could lead to a violation, i.e. $b$ could be set after $50ms$.
4. Based on formal requirements Test Cases and FMU Observers are generated by EP.
5. We instrument the system code to send required data to the generated monitors. The monitors evaluate the

[1]Details about FMU and FMI can be found at https://fmi-standard.org/.

specified properties and return an early verdict whether the system adheres to the system requirements. Note that it is preferable to monitor the system outline on a dedicated hardware to avoid impacts on the system.

⑥ Next, we run our system. First, we use the generated Test Cases. Then, simulations are used.

After each step, the artifacts are checked for violations, e.g. during execution the outcome of the FMU Observers are checked whether any requirements are violated.

## IV. Discussion

Currently, EP is able to realize the required functions. But its scalability on real-world example from aviation domain is unclear and needs to be addressed in future. First tests indicate that the formalization of textual requirements is easy to use. However, currently, one is limited to SUP which represent temporal trigger-action relationships. Still, based on experience, most of the functional requirements do actually fit in this pattern. For others, other languages exist like the stream-based specification language Lola [16] which was already used to monitor the status of unmanned aircraft systems during flight.

In future work, we have to investigate the trade-off of the approach in order to estimate the cost-benefit ratio between writing and formalizing requirements and the added value of better code at early stages, e.g. test cases and monitors. Clear metrics have to be found to estimate the benefit. Formalization overhead also incorporates mistakes during formalization. To which degree these kinds of mistakes are automatically detected by the consistency checks is also important.

During simulation, the use of generated monitors offer clear benefits like early feedback and avoidance of inline assertions. There, the main concern is the integration into the system setup, especially the system instrumentation. The monitors need to be able to run along the system while having a low impact on its execution. Figure 6 exemplary shows our current system instrumentation. We are not limited on passively reading databuses. We also instrumented the code to send information about the internal software status via UDP.

## V. Conclusion

We motivated a structured approach for prototyping. The approach is centered around the formalization of requirements. It allows to automatically check their consistency and to generate test cases and monitors. The approach can be implemented using open-source tools. As formal specification language, linear temporal logic (LTL) could be a candidate for which tools for model checking, runtime monitoring, and test generation do exist. We decided to use a commercial tool that combines all of these approaches. We showed how the integration into development could be done. Then, we discussed the barriers for an actual integration. In future, we plan to actually apply the approach and to report findings to come closer to answering the question whether formal methods pay off for prototyping.

```
# ———————— Monitor-side ————————
fmu = FMU(path)
fmu.instantiate(...)
mapping = {Variable -> (ID,Type)}
socket=openSocket(ip,port,mapping)
while(true):
   ID,data,evaluate=socket.rcv()
   fmu.setValue(ID,data)
   if evaluate:
     fmu.doStep(...)
     for key in mapping:
        print(key,fmu.getBooleanValue(key))
 ... Teardown
# ———————— System-side ————————
 ...    start up ... initialise socket
while(true):
   hgt_cmd,cur_hgt=getHgtCmd(),getCurHeight()
   obs = checkObstacleAbove()
   cmd_hgt = calculation(hgt_cmd,cur_hgt,obs)
   setCmdHeight(cmd_hgt)
   sendData(
   ['request_ascending','increase_hgt','obstacle']
   [ hgt_cmd > 0, cmd_hgt > 0, obstacle_above ] )
 ... Teardown
```

Fig. 6. Simplified code fragments for the data exchange between the instrumented system and the generated monitors.

## References


[1] RTCA, "DO-178C/ED-12C Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, 2011

[2] SAE International, "ARP 4754A Guidelines for Development of Civil Aircraft and Systems," SAE International, 2010

[3] Y. Falcone, K. Havelund, and G. Reger, "A tutorial on runtime verification," In Engineering Dependable Software Systems, 2013

[4] K. Y. Rozier, "From Siumlation to Runtime Verification and Back: Connecting Single-Run Verification Techniques," In SpringSim, 2019

[5] D. Phan, N. Paoletti, R. Grosu, N. Jansen, S. Smolka, and S. Stoller, "Neural Simplex Architecture," arXiv 1908.00528. 2019

[6] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "The Simplex Architecture for Safe Online Control System Upgrades," In Proc. 1998 American Control Conference Vol. 6.

[7] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," Proceedings of the IEEE International Symposium on Requirements Engineering, 1993

[8] Davis, Alan. "Just enough requirements management: where software development meets marketing." Addison-Wesley, 2013.

[9] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm, 'Integration of formal analysis into model-based software development process," Formal Methods for Industrial Critical Systems, 2008

[10] P. Skokovic, "Requirements-Based Testing Process in Practice," International Journal of Industrial Engineering and Management (IJIEM), Vol.1 No 4, 2010, pp. 155 - 161, ISSN: 2217-2661, 2010

[11] C. Torens, F.-M. Adolf, "Fail-Safe Systems from a UAS Guidance Perspective," In: Encyclopedia of Aerospace Engineering UAS. Wiley. ISBN 9780470686652

[12] T. Teige, T. Bienmüller, and H.J. Holberg, "Universal Pattern: Formalization, Testing, Coverage, Verification, and Test Case Generation for Safety-Critical Requirements," MBMV 2016: 6-9

[13] J.S. Becker, "Analyzing Consistency of Formal Requirements," ECEASST 76 (2018)

[14] T. Bienmüller, T. Teige, A. Eggers, and M. Stasch, "Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation," FM&MDD 2016

[15] H. Esen, M. Kneissl, A. Molin, S. vom Dorff, B. Böddeker, E. Möhlmann, U. Brockmeyer, T. Teige, G.G. Padilla, S. Kalisvaart, "Validation of Automated Valet Parking," Validation and Verification of Automated Systems, Springer, 2019

[16] F-M. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, and C. Torens, "Stream Runtime Monitoring on UAS," In Runtime Verification, 2017