From Requirements to Autonomous Flight: An Overview of the Monitoring ICAROUS Project

Aaron Dutle[1] Laura Titolo[2] Dimitra Giannakopoulou[3]
César Muñoz[1] Ivan Perez[2] Anastasia Mavridou[4]
Esther Conrad[1] Swee Balachandran[2] Thomas Pressburger[3]
Alwyn Goodloe[1]

[1]NASA Langley Research Center, Hampton, Virginia
[2]National Institute of Aerospace, Hampton, Virginia
[3]NASA Ames Research Center, Moffett Field, California
[4]KBR Inc. / NASA Ames Research Center, Moffett Field, California

{*aaron.m.dutle, cesar.a.munoz, esther.d.conrad, a.goodloe, laura.titolo, ivan.perezdominguez, sweewarman.balachandran, dimitra.giannakopoulou, anastasia.mavridou, tom.pressburger*}*@nasa.gov*

The Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS) is a software architecture incorporating a set of algorithms to enable autonomous operations of unmanned aircraft applications. This paper provides an overview of *Monitoring ICAROUS*, a project whose objective is to provide a formal approach to generating runtime monitors for autonomous systems from requirements written in a structured natural language. This approach integrates FRET, a formal requirement elicitation and authoring tool, and Copilot, a runtime verification framework. FRET is used to specify formal requirements in structured natural language. These requirements are translated into temporal logic formulae. Copilot is then used to generate executable runtime monitors from these temporal logic specifications. The generated monitors are directly integrated into ICAROUS to perform runtime verification during flight.

1 Introduction

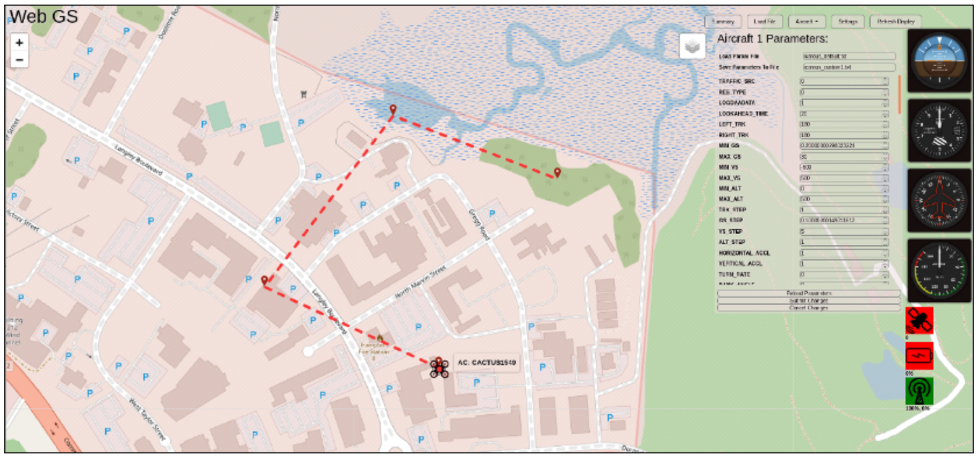
The Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS) [5] is a software architecture for enabling safe autonomous operation of unmanned aircraft systems (UAS) in the airspace. The primary goal of ICAROUS is to provide autonomy to enable beyond visual line of sight (BVLOS) missions for UAS without the need for constant human supervision/intervention. ICAROUS provides highly-assured functions to avoid stationary obstacles, maintain a safe distance from other users of the airspace, and compute resolution and recovery maneuvers.

Hardware and software verification via formal methods offers the highest assurance of safety available for such cyber-physical systems. While there have been considerable advances in creating industrial-scale formal methods (e.g., [6, 12, 23]), it is not yet practical to apply them to an entire complex system such as ICAROUS. Formal verification is generally carried out on a model of a system rather than the software itself, and so the properties verified may not hold if the model is inaccurate or if other faults make the system behave unpredictably. Moreover, while there has been much progress made in verification of neural networks in particular ([11, 13], increasingly autonomous systems employing machine learning and similar methods are challenging for formal verification.

Runtime verification (RV) [10] is a verification technique that has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. In RV, the system is monitored during execution, and property violations can be detected and acted upon during the mission. RV detects when properties are violated at runtime, so it cannot enforce the correct operation of a system, but is an improvement over testing alone, and can enhance testing by finding real cases of requirement violation. Copilot [17] is a runtime verification framework developed by NASA researchers and others.

While RV can be used to monitor and detect property violations, the actual properties to be monitored must be determined and specified externally. Such safety requirements are generally written by hand in natural language, which can lead to ambiguity as to their meaning or applicability. Additionally, when runtime verification is used as a key safety component of an autonomous system, having clearly specified requirements that are properly translated into *executable* monitors is critical. FRETISH [9] is a structured natural language developed by NASA to write unambiguous requirements. The associated tool, FRET (Formal Requirements Elicitation Tool), provides a framework to write, formalize and analyze requirements and automatically generate temporal logic formulae from them.

The *Monitoring ICAROUS* project, a work-in-progress joint effort at NASA, will demonstrate the integration of robust requirements-based runtime verification applied to an autonomous flight system for unmanned aircraft using FRET, Copilot, and ICAROUS. This project brings together work that the NASA formal methods team has been doing for many years on requirements elicitation and specification, runtime verification, and assured autonomous aircraft software.



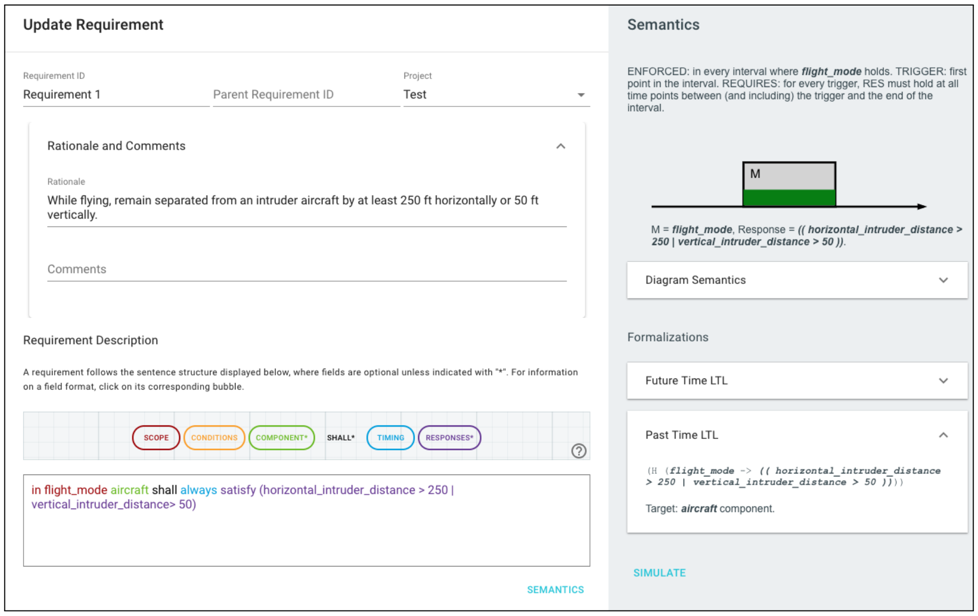
Aircraft 1 Parameters:

Load Param File	icarous_default.txt
Save Parameters To File	icarous_custom1.txt
TRAFFIC_SRC	0
RES_TYPE	0
LOGDAADATA	1
LOOKAHEAD_TIME	20
LEFT_TRK	180
RIGHT_TRK	180
MIN_GS	[illegible]
MAX_GS	50
MIN_VS	-500
MAX_VS	500
MIN_ALT	0
MAX_ALT	500
TRK_STEP	1
GS_STEP	[illegible]
VS_STEP	5
ALT_STEP	1
HORIZONTAL_ACCL	1
VERTICAL_ACCL	1
TURN_RATE	0

Figure 1: The current interface for setting parameters in ICAROUS.

The concept of operation for the integrated system is simple. Prior to the start of an autonomous flight with ICAROUS, an operator can set a collection of different mission and safety parameters (see Figure 1). For example, the detect-and-avoid module can be specified to avoid other aircraft by at least 250 ft horizontally and 50 ft vertically. With integrated monitoring, the related implicit requirements will become explicit ones, expressed in the structured natural language of FRETISH, and available to be viewed and edited by the user. In addition, a method for specifying custom requirements similar to the FRET interface will be available. These requirements expressed in FRETISH will be translated into Copilot's monitoring language, which will generate C code for the RV monitors. These monitors will be integrated into ICAROUS, which will use them to determine requirements violations. A violation of a monitor will alert the operator, who can use the information to return the aircraft to a safe state.

As a motivating example, the remainder of the paper will use the detect and avoid requirement *"Requirement 1: While flying, remain separated from an intruder aircraft by at least 250 ft horizontally or 50 ft vertically."* This safety property will be followed through the chain of tools employed, and illustrate



Requirement 1

Test

While flying, remain separated from an intruder aircraft by at least 250 ft horizontally or 50 ft vertically.

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

in flight_mode aircraft shall always satisfy (horizontal_intruder_distance > 250 | vertical_intruder_distance> 50)

ENFORCED: in every interval where ***flight_mode*** holds. TRIGGER: first point in the interval. REQUIRES: for every trigger, RES must hold at all time points between (and including) the trigger and the end of the interval.

M = ***flight_mode***, Response = ***((horizontal_intruder_distance > 250 | vertical_intruder_distance > 50))***.

```
(H (flight_mode -> (( horizontal_intruder_distance
> 250 | vertical_intruder_distance > 50 ))))
```

Target: ***aircraft*** component.

Figure 2: FRET editor, with the example requirement entered, and the *Semantics* pane visible.

the work to be done in the integration.

2 Tool Descriptions

FRET[1] is an open-source tool developed at NASA for writing, understanding, formalizing, and analyzing requirements. In practice, requirements are typically written in natural language, which is ambiguous and, consequently, not amenable to formal analysis. Since formal, mathematical notations are unintuitive, requirements in FRET are entered in a restricted natural language named FRETISH. FRET helps users write FRETISH requirements, both by providing grammar information and examples during editing, but also through textual and diagrammatic explanations to clarify subtle semantic issues.

Figure 2 illustrates FRET's requirements elicitation interface, with the example *Requirement 1* entered. The "Rationale and Comments" field holds the original text requirement; the "Requirement Description" field is where the FRETISH requirement is composed. Once a requirement is entered, the "Semantics" pane shows a text description of the FRETISH requirement, displays a "semantic diagram" showing a visual explanation of the requirement applicability over time, and provides translations from FRETISH to Metric Future- and Past-time linear temporal logic (LTL) [14].

A FRETISH requirement description is automatically parsed into six sequential fields; *scope, condition, component, shall, timing,* and *response*, with the FRET editor dynamically coloring the text corresponding to each field (Fig. 2). The mandatory *component* field specifies the component that the requirement applies to (**aircraft**). The *shall* keyword states that the component behavior must conform to the requirement. The *response* field currently is of the form *satisfy R*, where *R* is a non-temporal Boolean-valued expression (**horizontal_intruder_distance**$>$**250** $|$ **vertical_intruder_distance**$>$**50**). The (optional) *scope* field states that the requirement is only assessed during particular modes, for the example, in **flight_mode**. The (optional) Boolean expression field *condition* states that, within the specified mode,

[1]`https://github.com/NASA-SW-VnV/fret`

the requirement becomes relevant only from the point where the condition becomes true. The (optional) *timing* field specifies at which points the response must occur, in the example "always", meaning at all points in flight mode.

FRET automatically produces formulas in several formal languages, including Metric Future-time LTL and Past-time LTL. FRET also offers an interactive visualizer, showing temporal traces of each of the signals (variables) involved as well as the valuation of the requirement for each point in time.

Copilot[2] is an open-source runtime verification framework for real-time embedded systems. Copilot monitors are written in a compositional, stream-based language. The framework translates monitor specifications into C99 code with no dynamic memory allocation and executes with predictable memory and time, crucial in resource-constrained environments, embedded systems, and safety-critical systems.

The Copilot language has been designed to be high-level, easy to understand, and robust. To prevent errors in the RV system that could affect systems during a mission, the language uses advanced programming features to provide additional compile-time and runtime guarantees. For example, all arrays in Copilot have fixed length, which makes it possible for the system to detect, before the mission, some array accesses that would be out of bounds.

Copilot supports a number of logical formalisms for writing specifications including a bounded version of Future-time Linear Temporal Logic [20], Past-time Linear Temporal Logic (PTLTL) [15], and Metric Temporal Logic (MTL) [14]. Copilot also includes support libraries with functions such as majority vote, used to implement fault-tolerant monitors [19]. *Requirement 1* expressed as a PTLTL Copilot specification is as follows:

```
alwaysBeen (flightMode ==> (  horizontalIntruderDistance > 250
                           || verticalIntruderDistance   > 50 ))
```

ICAROUS[3] is an open-source software architecture developed at NASA to enable safety-centric autonomous aircraft missions. It is a service-oriented architecture, where service applications provide various capabilities such as path planning, sense and avoid, geofence containment, task planning, and more, through a publish-subscribe middleware. Applications are logically organized into conflict detectors, conflict resolvers, mission managers, and decision makers.

Conflict detectors are algorithms that check for imminent violation of constraints such as geofences, conflicts due to other vehicles in the airspace, deviations from mission flight plan, etc. These conflict detecting applications can also provide *tactical* resolutions, which provide a simple maneuver which will prevent the corresponding conflict. Conflict resolvers compute resolutions to prevent imminent violation of specified constraints. Resolvers may handle multiple conflicts simultaneously, and can provide *strategic* resolutions that are computed to prevent one or more constraint violations. A decision making application receives conflict information from monitors and triggers resolvers to compute resolutions for one or more conflicts. When resolving imminent constraint violation, outputs from mission applications are ignored. The mission is resumed once all conflicts are resolved.

These services are connected through the NASA core Flight System (cFS) middleware,[4] a platform-independent reusable software framework and a set of reusable software applications. The three key aspects to the cFS architecture are a dynamic run-time environment, layered software, and a component-based design. These key aspects make cFS suitable for reuse on any number of embedded software systems. The cFS middleware simplifies the flight software development process by providing the un-

[2] https://copilot-language.github.io/
[3] https://github.com/nasa/icarous
[4] https://cfs.gsfc.nasa.gov/

derlying infrastructure and hosting a runtime environment for development of project/mission specific applications.

3 Integration

The integration of the three systems described in Section 2 requires several steps in order to create a complete framework. The three major steps in the integration are as follows. The first step is developing a method for ICAROUS-specific requirements and safety properties to be specified in FRET. Next, the requirements expressed in FRETISH must be translated into Copilot. Finally, the monitors generated by Copilot must be integrated into ICAROUS in a usable way. Each of these integration steps are discussed in turn below, and the toolchain is depicted in Figure 3.

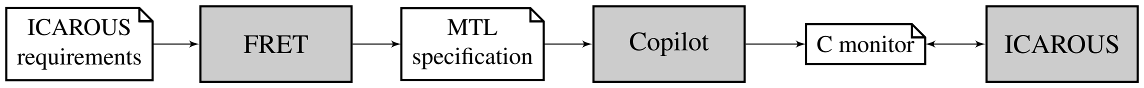


Figure 3: Toolchain to automatically generate monitors for ICAROUS.

3.1 ICAROUS interface to FRET

In order to facilitate the use of FRET for ICAROUS-specific requirements, several things need to be done. The first of these is a thorough accounting of all of the modes, systems, and signals available within ICAROUS for a monitor to access.

In FRET, the specification of safety properties is completely independent of the system that is being considered. This makes FRET very general and powerful in its ability to craft requirements, but makes it somewhat cumbersome to use in the specific setting of ICAROUS. The variables that FRET uses to refer to modes, systems, and signals in requirements are completely arbitrary, while being able to use such a requirement as a monitor means that each of these variables must correspond to an actual system or signal in ICAROUS. After each of these possible variables in ICAROUS is identified, along with its datatype information, FRET can be restricted to using only information that can be obtained by the system in specifying properties.

Another task to be completed is the generation of ICAROUS-specific templates for FRET. Many of the safety requirements that an autonomous flight system will be expected to follow exhibit similar patterns from flight to flight. For example, an autonomous operation may be required to stay below a certain altitude ceiling. More specific requirements such as *Requirement 1* are parametric requirements that a detect-and-avoid (DAA) system is expected to obey based on settings in the DAA system. For such requirements, a template can be given allowing the user to input particular values based on the mission, and the associated requirement added to those to be monitored. Additionally, many of the existing settings in an ICAROUS configuration carry with them implicit safety requirements. Setting the *max_altitude* parameter in the DAA system of ICAROUS can be interpreted as a requirement that the aircraft never goes above the value set. Parsing these files can allow for the *automatic* generation of a requirement from the associated template and the variable setting.

3.2 FRET to Copilot translation

The main integration step between FRET and Copilot is to take a requirement expressed in FRETISH, and translate it into a Copilot monitor. A prototype tool named Ogma is being developed to create Copilot monitors from languages such as FRETISH, SPEAR [7], and AGREE [4]. The tool can translate the Past-time LTL formulas FRET generates, so it can process any requirements specified in FRETISH. The resulting Copilot monitor can then be automatically translated into C99 code that checks that a corresponding property holds during runtime. This conversion should occur transparently: users specify FRETISH requirements, and automatically obtain C code compatible with ICAROUS.

3.3 Copilot Monitors in ICAROUS

The integration of Copilot-generated monitors into ICAROUS should be fairly straight-forward. Since ICAROUS is already a service-oriented publish/subscribe architecture, the main issue is subscribing and routing the appropriate signals to the monitors, and returning a signal to the user that indicates which properties have been violated. To facilitate this integration, the Ogma tool automatically generates a cFS application, responsible for subscribing to the appropriate ICAROUS applications, making data available to Copilot, and handle runtime violations reported by Copilot.

A technical difficulty in this approach (mentioned in Section 3.1) is that the RV monitors are generated *after* requirements are specified, and then this C code is integrated into ICAROUS. Currently, ICAROUS is installed on the aircraft once, and, generally, only settings are changed between flights. With new monitors generated for each flight, ICAROUS must be configured, new monitors generated, and then the system installed on the aircraft before flight. A partial solution would be to include parametric versions of common monitors, and have the parameters instantiated prior to flight. Alternatively, a utility for including monitoring code could be added, since the RV service would not change.

4 Conclusion

The work described here is still in-progress. Additional work is being conducted tangential to this integration. The FRETISH language and semantics are being specified in the Prototype Verification System (PVS) [16], to prove that the evaluation of a FRETISH statement is the same as the evaluation of the LTL statement produced by FRET for all possible finite and infinite traces. Currently, the verification of the translation is done through a systematic, rigorous testing framework for finite traces up to a certain length. An embedding of FRETISH in PVS would also allow for formal reasoning about models of a system such as ICAROUS with respect to specified requirements.

Related work on requirements specification, RV, and autonomous flight systems is omitted here. The interested reader is directed to the corresponding sections of [8, 9] for requirements, [2, 10, 17, 18] for runtime verification, and [1, 5] for autonomous flight systems. Work that has a similar flavor to the integration of these tools includes [3], where the R2U2 [22] engine is used to monitor an automated and intelligent UAS Traffic Management System for adherence to safety requirements during operation. The specifications are written in the Mission-time Linear Temporal Logic (MLTL) [21], an extension of MTL, in contrast with the present approach where the specifications are given using structured natural language.

The *Monitoring ICAROUS* project will allow a user to obtain relevant requirements automatically or defined using the structured natural language FRETISH, translate these requirements into runtime monitors using Copilot, and seamlessly integrate these monitors into the autonomous flight system ICAROUS.

The framework supports simple requirements specification and analysis, with robust runtime verification, while the translation and integration steps are performed in the background. Requirements-based runtime monitoring demonstrates a real-world application of formal methods to increase the safety assurance of complex automated systems.

References

[1] S. Balachandran, C. Muñoz, M. Consiglio, M. Feliú & A. Patel (2018): *Independent Configurable Architecture for Reliable Operation of Unmanned Systems with Distributed On-Board Services*. In: *Proceedings of the 37th Digital Avionics Systems Conference (DASC 2018)*, pp. 1–6, doi:10.1109/DASC.2018.8569752.

[2] E. Bartocci, Y. Falcone, A. Francalanza & G. Reger (2018): *Introduction to Runtime Verification*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *Lecture Notes in Computer Science* 10457, Springer, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[3] M. Cauwels, A. Hammer, B. Hertz, P. Jones & K. Y. Rozier (2020): *Integrating Runtime Verification into an Automated UAS Traffic Management System*. In: *International workshop on moDeling, vErification and Testing of dEpendable CriTical systems, DETECT 2020*, pp. 340–357, doi:10.1007/978-3-030-59155-7_26.

[4] D. D. Cofer, Gacek. A., S. P. Miller, M. W. Whalen, B. LaValley & L. Sha (2012): *Compositional Verification of Architectural Models*. In: *Proceedings of the 4th International NASA Formal Methods Symposium (NFM 2012)*, *Lecture Notes in Computer Science* 7226, Springer, pp. 126–140, doi:10.1007/978-3-642-28891-3_13.

[5] M. Consiglio, C. Muñoz, G. Hagen, A. Narkawicz & S. Balachandran (2016): *ICAROUS: Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems*. In: *Proceedings of the 35th Digital Avionics Systems Conference (DASC 2016)*, pp. 1–5, doi:10.1109/DASC.2016.7778033.

[6] Byron Cook (2018): *Formal Reasoning About the Security of Amazon Web Services*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 38–47, doi:10.1007/978-3-319-96145-3_3.

[7] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello & J. A. Davis (2017): *SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements*. In: *Proceedings of the 9th International NASA Formal Methods Symposium (NFM 2017)*, *Lecture Notes in Computer Science* 10227, pp. 420–426, doi:10.1007/978-3-319-57288-8_30.

[8] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann & N. Shi (2020): *Formal Requirements Elicitation with FRET*. In: *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020).*

[9] D. Giannakopoulou, T. Pressburger, A. Mavridou & J. Schumann (2020): *Generation of Formal Requirements from Structured Natural Language*. In: *26th International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ 2020*, *Lecture Notes in Computer Science* 12045, Springer, pp. 19–35, doi:10.1007/978-3-030-44429-7_2.

[10] K. Havelund & A. Goldberg (2008): *Verify Your Runs*, pp. 374–383. *Lecture Notes in Computer Science* 4171, Springer, doi:10.1007/978-3-540-69149-5_40.

[11] K. Julian & M. Kochenderfer (2019): *Guaranteeing Safety for Neural Network-Based Aircraft Collision Avoidance Systems*. *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pp. 1–10, doi:10.1109/DASC43569.2019.9081748.

[12] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber et al. (2009): *Replacing Testing with Formal Verification in Intel® CoreTM i7 Processor Execution Engine Validation*. In: *Computer Aided Verification*, Springer, pp. 414–429, doi:10.1007/978-3-642-02658-4_32.

[13] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer & C. Barrett (2019): *The Marabou Framework for Verification and Analysis of Deep Neural Networks*. In I. Dillig & S. Tasiran, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 443–452, doi:10.1007/978-3-030-25540-4_26.

[14] R. Koymans (1990): *Specifying Real-time Properties with Metric Temporal Logic*. *Real-Time Syst.* 2(4), pp. 255–299, doi:10.1007/BF01995674.

[15] F. Laroussinie, N. Markey & P. Schnoebelen (2002): *Temporal Logic with Forgettable Past*. In: *LICS02: Proceeding of Logic in Computer Science 2002*, IEEE Computer Society Press, pp. 383–392, doi:10.1109/LICS.2002.1029846.

[16] S. Owre, J. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In: *Proceeding of the 11th International Conference on Automated Deduction (CADE)*, *Lecture Notes in Artificial Intelligence* 607, Springer, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[17] I. Perez, F. Dedden & A. Goodloe (2020): *Copilot 3*. Technical Report NASA/TM2020220587, NASA Langley Research Center, doi:10.13140/RG.2.2.35163.80163.

[18] L. Pike, A. Goodloe, R. Morisset & S. Niller (2010): *Copilot: A Hard Real-Time Runtime Monitor*. In: *Proceedings of the First International Conference on Runtime Verification (RV 2010)*, *Lecture Notes in Computer Science* 6418, Springer, pp. 345–359, doi:10.1007/978-3-642-16612-9_26.

[19] L. Pike, N. Wegmann, S. Niller & A. Goodloe (2013): *Copilot: monitoring embedded systems*. *Innovations in Systems and Software Engineering* 9(4), pp. 235–255, doi:10.1007/s11334-013-0223-x.

[20] A. Pnueli (1977): *The Temporal Logic of Programs*. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, IEEE Computer Society, Washington, DC, USA, pp. 46–57, doi:10.1109/SFCS.1977.32.

[21] T. Reinbacher, K. Y. Rozier & J. Schumann (2014): *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*. In: *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, *Lecture Notes in Computer Science* 8413, Springer, pp. 357–372, doi:10.1007/978-3-642-54862-8_24.

[22] J. Schumann, P. Moosbrugger & K. Y. Rozier (2015): *R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems*. In: *Proceedings of the 6th International Conference on Runtime Verification (RV 2015)*, *Lecture Notes in Computer Science* 9333, Springer, pp. 233–249, doi:10.1007/978-3-319-23820-3_15.

[23] J. Souyris, V. Wiels, D. Delmas & H. Delseny (2009): *Formal Verification of Avionics Software Products*. In: *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, Springer-Verlag, Berlin, Heidelberg, p. 532546, doi:10.1007/978-3-642-05089-3_34.