# Development of a safe CPS component: the hybrid parachute, a remote termination add-on improving safety of UAS

Laurent Ciarletta[1], Loïc Fejoz[2], Adrien Guenard[3] and Nicolas Navet[4]

[1]Loria, MinesNancy, University of Lorraine, INRIA project-team Madynes, France
[2]RealTime-at-Work, France
[3]ALERION, France
[4]University of Luxembourg, Luxembourg

**Abstract**

The use of Unmanned Aerial Systems (UAS) can be leveraged in many application domains ranging from agriculture to industry, opening up a wealth of new possibilities. However, UAS obviously raise important safety concerns and the use of the techniques, processes and standards developed for the aeronautic industry is not a feasible solution for most UAS. There is a need to bring in novel and pragmatic solutions to develop provably safe UAS in a time and cost-affordable manner. This paper reports on the development of a smart parachute which provides a safe-crash (termination) solution for UAS, one of the core safety requirements which can be complemented by other safety components in an incremental manner. The requirements elicitation phase, the design and partial verification of the termination system has been carried out using CPAL, a lightweight model-based design environment for embedded systems. The study illustrates on a specific requirement of the system how simulation and fault-injection on models can be used to provide evidence that the parachute system meets its design objectives.

## 1 Introduction

### 1.1 Context of the study

Drones or Unmanned Aerial Vehicles (UAV) or Systems (UAS) have been increasingly spotted on the civilian radars. Everywhere on the news, they can be seen as business opportunities in many fields from agriculture to industry, as pure entertainment devices (coming from the RC world) or as ethical and sociological subjects of interest or concerns (automated aircrafts or vehicles can be used to carry weapons, or as privacy invading tools). Having hundreds or thousands of mostly autonomous UAVs flying in rural but also urban airspaces raises important safety concerns (see [13]). One the one hand, applying or enforcing the exact same techniques and guidelines used in the aeronautic industry for the certification of small to middle size (and weight) UAVs is not a reasonable solution as of today. But on the other hand, if an RPAS (Remotely Piloted Aircraft System) / UAV industry emerges where such vehicles become ubiquitous as forecast, safety becomes a main concern for the provider of hardware and software, the operator of the system, the insurance companies or the regulation bodies. Indeed, the actual cost of certification, if applied to every part, every modification for every UAV, cannot deliver solutions even for professional systems that cost only a few thousands of euros, and could potentially be rapidly modified and updated in their uses and configurations. At least this is what is targeted for general public and professional use.

We believe that this is an opportunity to bring novel and pragmatic solutions by incrementally adding safety to the system. Beginning with a smaller set of safety functions (termination for example), and targeted subsystems the industry and legislator could increasingly extend the safety requirement to all functions and parts of the entire UAS. ALERION is promoting a design framework to build adaptive and tailor-made UAS by the integration of (secure and) provably safe Cyber Physical components. Building on the experience of the participation to the final of the UAV Outback "Search and Rescue" challenge (see `http://uavchallenge.org/search-and-rescue/`), which implied the fulfillment of a set of safety requirements, ALERION is developing in partnership with RTaW a Smart Hybrid Parachute system. This system is an all in one (*i.e.*, hardware and software) add-on termination system for any UAV. It

works independently of the UAV's normal operation and can be triggered either by the operator through a safe, secure and dedicated communication channel or upon the detection of specific error conditions (e.g., hardware or software failures, system out of the authorized flight envelope, communication problems).

### 1.2 Contribution of this study

The main goal of this study is to demonstrate that such safety components can be developed according to a safety process in a time and cost-affordable manner, and that they would add the minimum level of safety requirements to allow a safe-crash (termination) solution for UAS. We describe design decisions and return of experience throughout the development process, with a focus on requirements and design phases.

The requirements elicitation phase, the design, simulation, and verification of the termination system has been carried out using CPAL [9], a lightweight model-based design environment for embedded systems jointly developed by RTaW and the University of Luxembourg. The complete set of requirements, the CPAL development environment (see `http://www.designcps.com`) and the CPAL models are made freely available for all uses. Though the verification stage is by no means complete, the first tests carried out using simulation and fault-injection on models suggest that the parachute system can meet its design requirements and provide a cost-effective solution to increase the safety of UAVs.

Finally, we would like to emphasize that if all the classical requirements from aeronautics should be considered, UAVs offer an opportunity to streamline the process of verification without giving away too many of the good practices of today's aeronautical industry.

## 2 Smart Parachute System : Remote Safety for Autonomous Vehicle Application

Very promising novel applications and markets are foreseen with UAS. But as far as micro drones are concerned, it is not possible today to enforce the same quality and safety mechanism as in private or commercial aviation[1]. Paying even a few tens of thousands euros to certify an autopilot, each and every time a new version is rolled out, can hardly work for devices that cost between a hundred and a few thousands euros. These systems are however already able to carry significant loads and represent a physical danger to the population or the institutional and industrial infrastructures.

Fully autonomous systems are not allowed so far and a human must remain in control during the operations. For now, the regulation tends to focus on the responsibility of the drone sector actors with a good part on the operator or pilot, backed by its employer. This is why in Europe it is mainly referred as RPAS (Remotely Piloted Aircraft Systems). But the systems are a lot closer to improved RC systems than to downsized aircrafts. The recreational use is now also under scrutiny. The RC world used to be a microcosm of very dedicated fans, while the "drone" is more of a mainstream activity. Very importantly, it is not clear who will be responsible in case of an accident: many non-certified hardware and software failures could potentially be invoked and it would be difficult to pinpoint the exact cause of the problem. Eventually, more autonomous UAS will require achieving and demonstrating higher safety levels.

### 2.1 Minimal safety for general public UAVs with our safe termination system

When dealing with autonomous systems, be they robots or autonomous vehicle, it is most often required to add some "emergency shutdown" mechanism, typically called the "red button". In case of UAS, we propose to have a safe-crash vision of safety: being able to terminate a flight when some errors conditions are observed (control loss, immediate danger etc.). This has been well stated for example in the rules of the UAV challenge "Outback Joe" [3]. The simplest of the requirements is therefore to have a termination procedure enforced on the UAV which can be triggered by the operator. Another general requirement is to have the termination procedure engaged when for example the radio contact is lost with the operator / pilot. This is a way to ensure no "out of sight" flight.

[1] The currently ongoing CAP 2018 FUI French collaborative project aims at developing the first autopilot for autonomous drones that can be certified according to DO178-C, see [14]. To the best of our knowledge, there is at the time of writing no publicly available outcome of the project.

Generally with modern autopilots for UAVs you can program that type of behavior and have some switches on your remote that actually triggers the termination or at least the return to base procedure. Most of the autopilots are however either black boxes or open source software that are constantly evolving with new features but are rarely checked as far as software correctness is concerned. Therefore, there is a high probability that native safety "mechanisms" will not be available in software in case of errors at runtime. This is the reason why ALERION has decided to develop a Smart Parachute System that works as an add-on to any already existing system. This Smart Parachute constitutes the core safety mechanism of the UAV and it has to be developed with strong safety requirements to ensure safe-crash.

Figure 1: From simulation to field test ("red button" prototype on the right).

At ALERION, we have been using several ways of providing a certain level of quality with our development tool-chain. We are using modeling environment, simulation, hardware-in-the-loop and software-in-the-loop to develop, taking into account safety concerns and accurate physical models [1,2]. We combine ROS (Robot Operating System), with MAVLink (protocol) ready devices and are using open-source autopilots. Even though the underlying OS may be "real-time" (e.g., NuttX RTOS for Pixhawk), from our experience in real flights, and analysis of parts of the code, the actual autopilot function cannot always be fully trusted. Our view is that the focus of the current developments in the UAV community is more on functionalities and ease of development rather than safety of the solutions.

## 2.2 Smart parachute requirements: methodology and tool support

The design of critical embedded systems is usually carried out by large OEMs using well-established processes. It is also known that the certification process costs a significant fraction of the overall design costs. Yet, we believe that these processes can be scaled down, made faster, and adapted to the design of systems that are smaller and cheaper than aircrafts, satellites or power plants. One way to achieve this is through the development of the appropriate standards, such as the ISO 29110 [5] meant for entities and services with less than 25 people, inspired by the bigger ISO/IEC/IEEE 15288. The designer needs also to be provided with the tools that will guide him along the process, and help him learn and master the state-of-the-art practices.

### 2.2.1 Support for requirements elicitation

The successful design of a system starts from a good specification. Yet writing a complete and coherent specification is hard, and systems engineering skills require time and experience. To help designers with requirements elicitation, RTaW provides ReqLab [5], an online requirements editor conceived to be as easy to use as a spreadsheet, for instance to edit and organize requirements, but also offering the possibility of more advanced mandatory features like requirements traceability.

If, ideally, one should be able to share a model of the system as a specification, typically a SysML model, model-based specification only is not always feasible. ReqLab provides thus features to automatically generate documents from the underlying requirements by extracting text or creating diagrams.

Finally, ReqLab tries to guide the user. On purpose, in comparison to other requirements editors, ReqLab comes with less features and a reduced degree of control. For instance, the number of links for traceability is limited to refinement. Another illustration is the usage of tags to separate between what is a "real" requirement and what is a goal (as in KAOS/GORE [6]). As a result, the user has only a few concepts to understand to start using the software.

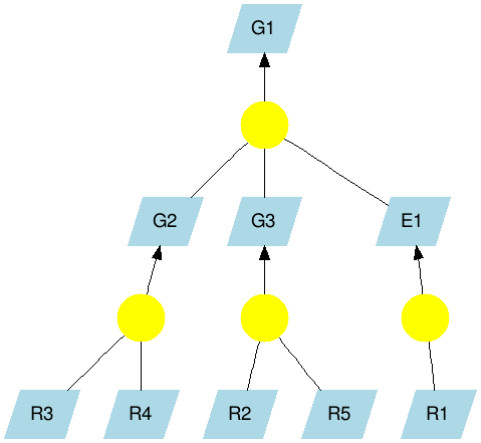



Figure 2: Generated synthesis document in KAOS/GORE notation.

The user is encouraged to start by the mission of the system and refines it with everything that is mandatory to achieve it. Obviously those are not yet requirements per se. Hence they are tagged as "Goal". The user can write a document that extracts some of those requirements for a more understandable presentation. In the case of the Smart Parachute, the list of requirements includes:

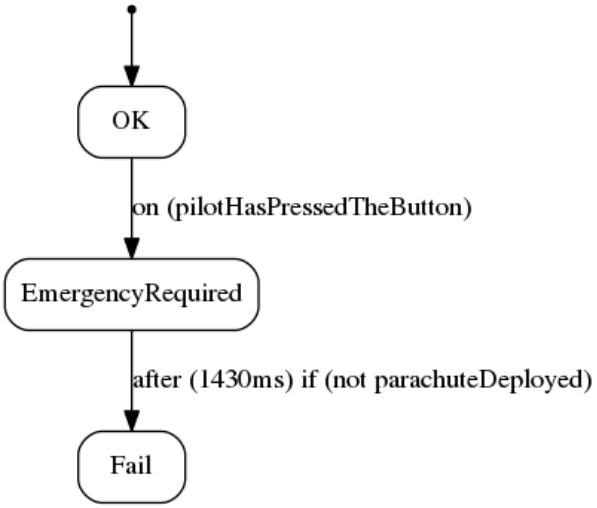
- G1 Reduce goods damage.
- G2 Remote safety procedure shall deploy a parachute.
- G3 When communication link loss is detected, the remote safety procedure shall be engaged.
- E1 The pilot shall engage the remote safety procedure every time a hardware failure occurs, or when an emergency is going to happen.
- [R1] Every time the pilot shall be able to manually start the remote safety process.
- [R2] The remote system shall engage the remote safety process if it has received no message for 1s.
- [R3] The safety process shall turn the propellers off before deploying the parachute.
- [R4] Once the safety process engaged, the parachute shall be deployed in less that 1.43s.
- [R5] The ground control system shall send a message to the remote system every 100ms.

The value 1.43 second in requirement R4 has to be derived from a set of parameters such as the ground speed target, the minimum acceptable flight altitude, the weight of the UAS and the characteristics of the parachute (opening time, lift, etc). The maximum speed with with the UAS may hit the ground may be specified by local regulation documents (see [12] for France). A more complete list of requirements for the smart parachute is available at `https://www.requirements.fr/api/v1/docfrags/ertss2016-parachute~specification`.

### 2.2.2 Executable requirements

The designer can then refine deeper his list of requirements down to a specification, *i.e.* a list of requirements that are SMART (Specific, Measurable, Assignable, Relevant/Realistic and Testable/Timebound). The fulfillment of SMART requirements can be verified in a dedicated CPAL task. CPAL natively support finite state machines, more precisely mode-automata [8], to describe the logic of the tasks. Along with CPAL simulation capability (faster than real-time), we can easily write tasks that check that a property holds during the simulation of a certain scenario of execution, or during the execution of the real system. In addition, knowing that we will execute requirements force to write SMART requirements. For instance, requirement R4 for the smart parachute could be verified with the code shown in Figure 3. It is a classical implementation of an observer automaton for temporal logic

properties. The designer can write tasks to stimulate the core design along with their observers to check the fulfillment of the requirements. Both design and validation of the software will be further explained in the following sections.




```
processdef R40bserver (
  in bool : pilotHasPressedTheButton,
  in bool : parachuteDeployed)
{
  state OK {
  }
  on (pilotHasPressedTheButton)
    to EmergencyRequired;
  state EmergencyRequired {
  }
  after (1430ms) if (not parachuteDeployed)
    to Fail;
  state Fail {
    /* println("R4 FAILED"); */
    assert(false);
  }
}
```

Figure 3: Formalisation of requirement R4: code on the right and visual representation on the left.

To counteract the natural tendency of focusing on functional aspects only, CPAL provides means to define precisely the timing behaviour of the system. For instance, CPAL language enforce the organisation in tasks with well-defined activation properties (e.g., periodic tasks with offsets). Moreover the CPAL editor show the Gantt chart of the task activations. This helps the designer verify that the timing behavior does not jeopardize the functional correctness of the system (e.g: excessive jitters, deadline misses, data are produced after being used, etc). This possibilities can be leveraged to verify requirements involving timing properties.

# 3 Software design and implementation

The system is really made up of two parts. One is called the transmitter, the other the receiver as seen in Fig. 1. The transmitter will is to be used by the pilot in case of emergency. The receiver will be embedded in the drone, and inserted between the RC receiver and the autopilot, so has to be able to turn the motor off and release the parachute. The CPAL code of the smart parachute system is available at `http://www.designcps.com/wp-content/uploads/ertss2016.zip`.

## 3.1 Functional architecture on sender and receiver side

The software architecture on the sender and receiver sides are quite similar as shown in Fig. 4 and 5. Both have in common 3 tasks (rounded rectangles):

- a task to manage the current global mode (`tm_modeTask` and `rcp_modeTask`),
- another to manage the user interface, *i.e.* LEDs in this implementation (`tm_uiTask` and `rcp_uiTask`),
- and a task dedicated to the wireless communication (Xbee in the prototype) (`tm_xbeeTask` and `rcp_xbeeTask`).

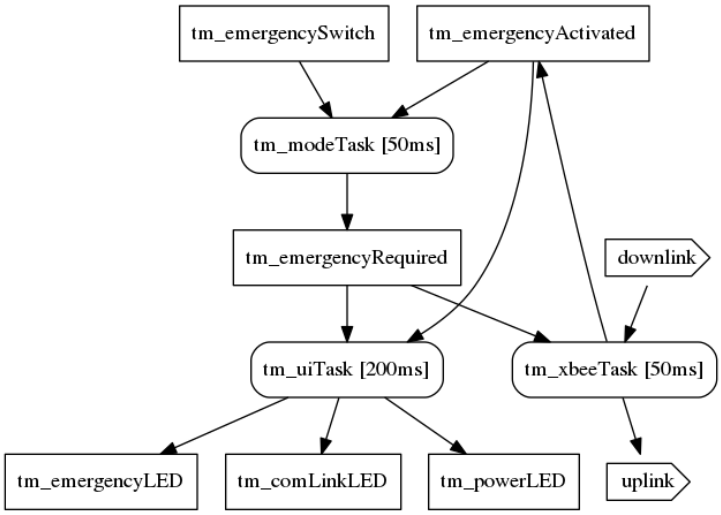



Figure 4: Software architecture of the transmitter: three tasks (rounded rectangles) `tm_modeTask`, `tm_uiTask`, and `tm_xbeeTask`, and their mean of communication (Rectangle and cds shape). `uplink` and `downlink` are channels of communication between transmitter and receiver, while the rectangles are global variables (screenshot from the CPAL editor).

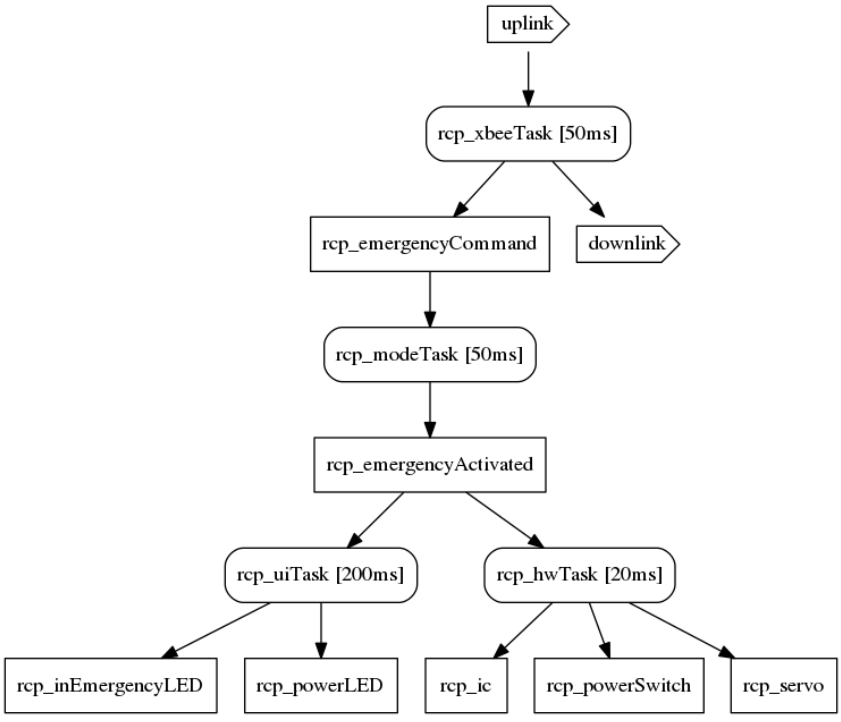



Figure 5: Software architecture of the receiver: Same architecture as the transmitter except one specific task `rcp_hwTask` to handle specifically the servo and electronic switches.

All rectangles in the Figures 4 and 5 are kinds of global variables used to share state information between tasks, referred to as *processes* in CPAL. The two others, `uplink` and `downlink`, are FIFO queues. In simulation, they are abstract Xbee messages while in the prototype implementation they are queues of characters as both Xbee modules are configured in transparent mode (*i.e.*, they implement a serial connection over-the-air, just like a wire would do). In the final prototype, they should be configured as Xbee frames so as to benefit from services of the Xbee protocol (e.g., "keep-alive" messages).

The receiving module executes another process, named `rcp_hawTask`, to control the ESCs (Electronic Speed Control), that reduces the motor's speed through a PWM signal command, and to trigger the opening parachute. Indeed requirement R3 requires to cut-off the motor before releasing the parachute. To do so, one need to by-pass the autopilot PWM commands with the `rcp_ic` electronic switch, send a zero command to the ESC via the `rcp_powerSwitch`, and finally release the parachute by setting the PWM `rcp_servo`.

## 3.2 Parachute deployment sequence

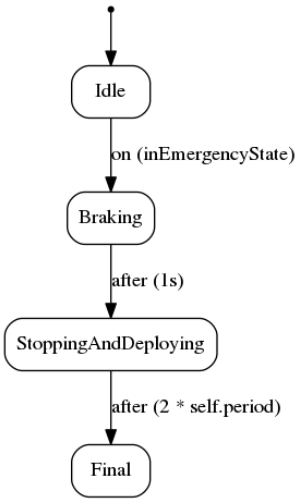



Figure 6: Logical sequence before parachute deployment.

The parachute deployment sequence, or any sequence with states and transitions between states, is easily described in CPAL. The CPAL editor also automatically displays the corresponding automaton graphically as shown on Figure 6. As seen on the figure, we let the motors brakes for 1s in order to delay the parachute deployment so that its strings do not get entangled in the moving propellers.

CPAL is not only a language but an execution platform. Indeed, with the same source code, one can interact physically with the GPIOs of a Raspberry Pi or a Freescale FRDM-K64F board, or with device drivers on an embedded Linux. Programming a GPIO, a PWM command, an analog-to-digital converter, etc, is simplified for the programmer since the low-level interactions with the hardware are performed by the interpreter. At run-time, the CPAL model is interpreted by an execution engine which relies on principles of similar systems deployed in interlocking systems, for instance at SNCF [7], where the hardware interpreter guarantees the semantics of the execution.

In terms of scheduling, the processes in charge of the user interface (`tm_uiTask` and `rcp_uiTask`) can be run at slower rate since they are meant to inform the user, for which a period of 200ms is sufficient. The communication tasks have a larger execution times than the other tasks because they are interacting with the Xbee module on the serial bus. It is crucial nonetheless that these communication tasks are able to react quickly, they are thus assigned an execution period of 50ms. The mode management task are on both data-flow, and, for this reason, is given a period of 50ms too.

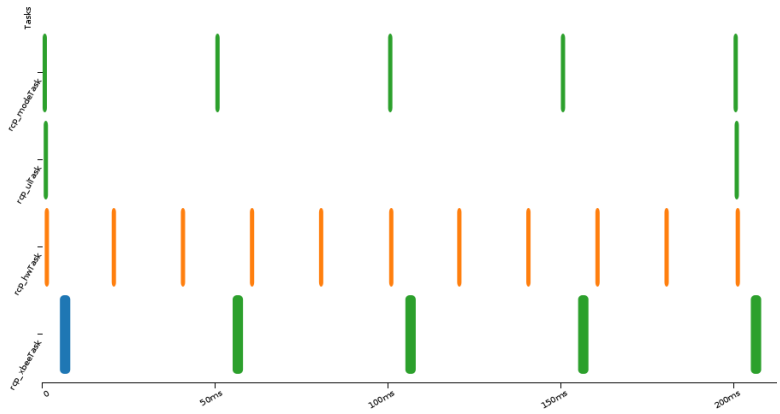



Figure 7: Gantt chart of the tasks activations on the receiver, from top to bottom: `rcp_modeTask`, `rcp_uiTask`, `rcp_hwTask` and `rcp_xbeeTask`. The color of a bar indicates the current state the task is in and the width of a bar indicates the task execution time.

The CPAL editor provides a timing diagram of the activation of all tasks, as shown on Figure 7 for the receiver. Design decisions can be taken by analyzing the timing behavior of the tasks on this Gantt diagram. For instance, in order to make the Xbee task really periodic, *i.e.* without any jitter, we have set an initial offset of 5ms for this task which will then not be delayed anymore by the other tasks. On the other hand, one observes that the `rcp_hwTask` is regularly slightly delayed by the `rcp_modeTask` and `rcp_uiTask`.

# 4 Verification of system correctness

## 4.1 Verification in nominal mode

The same CPAL source code can be run in real-time mode, where the execution of the code follows the timing specifications, typically tasks' periods, and it can also be run as fast as possible, in simulation mode. This latter mode enables to explore more trajectories of the system. Obviously such a verification by simulation will not be exhaustive, yet it is powerful in our experience (see [11]) and enforces good practices like formalizing requirements. For instance, we have included to the code on the receiving side the specification of requirement R4 and have checked by simulation of the model that the requirement holds.

```
@cpal:time {
  rcp_modeTask.wcet = 300us;
  rcp_uiTask.wcet = 300us;
  rcp_xbeeTask.wcet = 500us;
  rcp_hwTask.wcet = 500us;

  if (true) {
    rcp_modeTask.execution_time = rcp_modeTask.wcet;
    rcp_uiTask.execution_time = rcp_uiTask.wcet;
    rcp_xbeeTask.execution_time = rcp_xbeeTask.wcet;
    rcp_hwTask.execution_time = rcp_hwTask.wcet;
  } else {
    rcp_modeTask.execution_time = time64.rand_uniform( 2 * rcp_modeTask.wcet / 3, rcp_modeTask.wcet);
    rcp_uiTask.execution_time = time64.rand_uniform( 2 * rcp_uiTask.wcet / 3, rcp_uiTask.wcet);
    rcp_xbeeTask.execution_time = time64.rand_uniform( 2 * rcp_xbeeTask.wcet / 3, rcp_xbeeTask.wcet);
    rcp_hwTask.execution_time = time64.rand_uniform(2 * rcp_hwTask.wcet / 3, rcp_hwTask.wcet);
  }
}
```

Figure 8: Execution time annotations of tasks. The `execution_time` attribute of a task is used in simulation. Changing `true` to `false` in the code would simulate an execution time obeying a uniform distribution in the interval 2/3 of the WCET to the WCET, instead of always simulating the WCET.

CPAL offers support to express timing properties, which are often equally important as the functional behavior in real-time systems. For instance, it is possible to add timing annotations to a program, as an

example is shown in Figure 8. With such annotations, typically obtained by on-target measurements, it is possible to simulate the effects of timing behaviors, typically the execution times of the tasks, or to perform a worst-case schedulability analysis (see [10] for the schedulability analysis of CPAL tasks).

In our context, different timing behaviors can change the worst-case delay until the complete deployment of the parachute, *i.e.* the delay between when the user pushes the red button and when the parachute is fully deployed. This delay does not only depend on the opening sequence, but also scheduling of the tasks, and the time needed to transmit the message. All this can be simulated in CPAL on the basis of the same functional model.

## 4.2 Model-based fault-injection

It is needed during the design of many systems with dependability constraints to study their resilience to errors and faults that can happen at run-time. A powerful technique for this purpose is fault-injection which can be done on models or on the actual system. Here, we are interested in studying the impact of message losses, which are likely to happen with wireless communication and thus pose a threat to the correct functioning of the smart parachute. We can for instance introduce between the receiver and transmitter tasks a CPAL process simulating a faulty network, or simulate the fact that the first emergency message is loss. What we have done here, is simulate the random loss of up-link messages depending on a network quality ratio, and analyze the effect on the fulfillment of requirement R4 about the latency in deploying of the parachute.

We conducted simulations with a network quality ratio varying from 50% to 100% by step of 10% (5000 simulations per step). A network quality ratio of 50% means here that on average 50% of the messages are successfully transmitted. For each simulation, we have counted the number of times requirement R4 is satisfied. We have also recorded the minimum, average, and maximum time needed by the parachute to be deployed. All these data are presented in Figure 9.

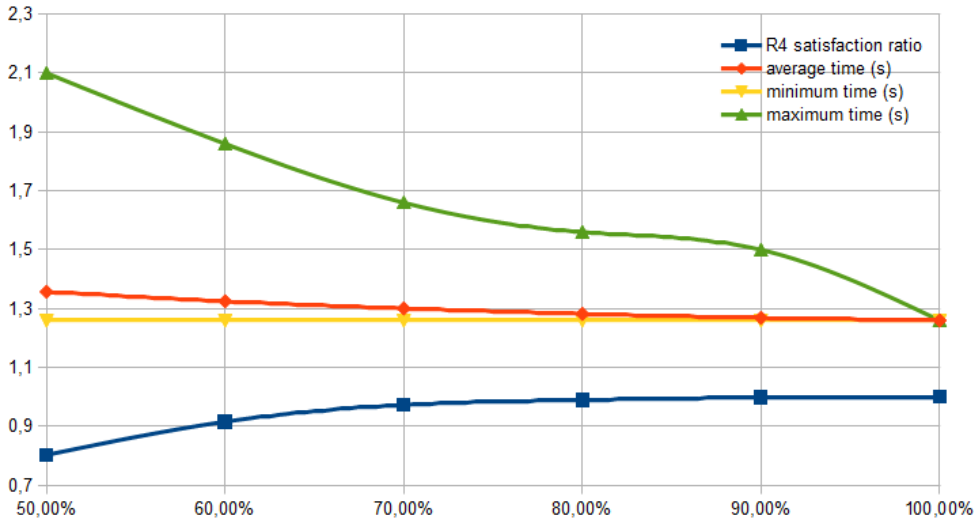



Figure 9: Time for the parachute to deploy (in seconds) and satisfaction of requirement R4 versus network quality ratio.

What we observe for instance, is that even with a 60% up-link network quality, R4 is satisfied 90% of the time. But the maximum deployment time can be then nearly half-more. It means that the parachute will have less time to decrease the speed of the UAV, and that the UAV will fall with greater energy on the ground. Obviously, the minimum time is constant and correspond to the critical path of the data-flow. The average time is not that different because once the emergency has been required, the transmitter kept sending an emergency command until it receives the acknowledgment that the sequence of deployment has been be triggered. Yet, without simulation, it would be difficult to derive the worst-case situation.

# 5 Conclusion

The paper reports on the development of a safe termination add-on component for UAS. Such a safety component improves the safety in the usual situation where the autopilot cannot be fully trusted. The component is developed in the CPAL language which has been designed to provide the right language abstractions to develop such embedded systems with dependability constraints. The use of model-interpretation makes it easier to verify the correctness of the system since the logic of the application is decoupled from the run-time services and written in a high-level language. The ability to easily express requirements in CPAL and verify them in simulation mode or real-time execution mode is also a powerful technique in our opinion.

The list of requirements of the termination system can be further refined, and the models extended accordingly. This ongoing work is being conducted on the basis of the experience gained on a prototype we are developing. The verification of the system correctness requires further work too, a question that needs to be investigated is whether the use of simulation alone can provide the verification coverage needed by such systems.

# References


1. Ciarletta L., Guénard A. , "The AETOURNOS project: Using a flock of UAVs as a Cyber Physical System and platform for application-driven research", EmSens 2012.
2. Ciarletta L., Guénard A., Presse Y., Galtier V., Song Y.Q, Ponsard J.C., Abarkane S., Theilliol D., "Simulation and Platform Tools to develop safe flock of UAVs: a CPS Application-Driven Research". In International Conference on Unmanned Aircraft Systems, ICUAS 2014.
3. UAV Challenge, "Search and Rescue Challenge - Mission, rules and judging criteria", version 1.4, August 2014.
4. Public site of the ISO Working Group mandated to develop ISO/IEC 29110, `http://profs.etsmtl.ca/claporte/English/VSE/index.html`, retrieved 2015/10/21.
5. RTaW-ReqLab requirements editor, `https://www.requirements.fr`, 2015.
6. Van Lamsweerde A., "Requirements Engineering - From System Goals to UML Models to Software Specifications", ISBN 0470012706, 2009.
7. Antoni M., "Formal validation method and tools for computerized interlocking system", FM 2012, Industry day, slides available at `http://fm2012.cnam.fr/fm2012/ID2012-Marc-Antoni.pdf`.
8. Maraninchi F., Rémond Y., "Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems", Science of Computer Programming, Elsevier, n°46, pp. 219-254, 2003.
9. Navet N., Fejoz L., Havet L., Altmeyer S, "Lean Model-Driven Development through Model-Interpretation: the CPAL design flow", to appear at ERTS2016. Preliminary version available as technical report from the University of Luxembourg at `http://hdl.handle.net/10993/22279`.
10. Altmeyer S., Navet N., "The case for FIFO scheduling", technical report from the University of Luxembourg, to appear, 2015.
11. Altmeyer S., Navet N., Fejoz L., "Using CPAL to model and validate the timing behaviour of embedded systems", WATERS Workshop, July 2015. Available at `http://hdl.handle.net/10993/21250`.
12. "Arrêté du 11 avril 2012 relatif à la conception des aéronefs civils", Annexe2-2.2.6, 2012. Available at `http://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000025834953&dateTexte=20151022`
13. Clothiera R.,Williams B., Fulton N., "Structuring the safety case for unmanned aircraft system operations in non-segregated airspace", Safety Science, vol.79, pages 213–228, November 2015.
14. "Sogilis, le projet CAP 2018 retenu au FUI20", `http://www.aerospace-cluster.fr/news-entreprises/sogilis-le-projet-cap-2018-retenur-au-fui20/`, Retrieved November 13, 2015.