Early Detection of Design Errors in the Life Cycle of Unmanned Aerial Vehicles
A SysML Approach

Pierre de Saqui-Sannes[1], Ludovic Apvrille[2], Rob Vingerhoeds[1]

(1) ISAE-SUPAERO, Université de Toulouse, France
(2) LTCI, TelecomParisTech, Université Paris Saclay, France

pdss@isae-supaero.fr, ludovic.apvrille@telecom-paristech.fr, rob.vingerhoeds@isae-supaero.fr

Abstract

The widespread of Unmanned Aerial Vehicles (UAVs) in various application domains has questioned the design methods used by UAV manufacturers. Migration from document centric approaches to Model-Based ones has stimulated research work on modeling languages and tools that reduce cost development and time to market. Among the various benefits one may expect from using a Model-Based System Engineering approach, the paper essentially considers a model as a reference for early detection of design errors in the life cycle of UAVs. The paper proposes designers to model the UAV in SysML and to use the free software TTool for safety analysis. TTool includes a SysML model editor, a model simulator and formal verification modules that rely safety analysis on mathematics rather than chance. The method associated with SysML and TTool is applied to a UAV in charge of taking pictures.

1. Introduction

The purpose of this paper is to illustrate a model-based systems engineering approach to the development of a UAV (Unmanned Aerial Vehicle) and the early detection of the potential design errors for such vehicles. Using a systems engineering approach, a complete view on the system (in this case a UAV) and its environment is taken. A system can be defined as "… an integrated set of elements, subsystems or assemblies that accomplish a defined objective. These elements include products (hardware, software, firmware), process, people, information, techniques, facilities, services and other support elements." (INCOSE Handbook V4, July 2015).

A system has a purpose, a mission to fulfill, often to provide a solution to a (business) problem. For example, a UAV can have a mission to transport packages from one location to another to another via air-transportation in an autonomous manner without bringing in danger the load (the packages) or other users of the infrastructure (in air: other aeronautical vehicles, on ground: houses, buildings, people, cars, etc.).

A structured approach to system design and engineering is needed to guide the complete life cycle of a changing system. Such an approach begins with requirements that express the wants and needs of stakeholders, in addition to feasibility studies, a concept of operations, known regulations and other sources of information. In successive stages, requirements and designs are expanded and refined through numerous iterations from top-level designs to detailed and operational designs. Design options are often not independent - they are interconnected in many ways, with each other and with the environment in which they are located. The components or subsystems of the system, their interconnections and boundaries, and their interactions with their environment are not accidental but result from deliberate and often multidisciplinary design and engineering.

A clear, unambiguous mission statement is key for the development of the system, and must be clearly stated by stakeholders. Stakeholders are those persons / entities / organisations / … that have a stake in the system. In the above example of an autonomous UAV, one can think of the owners of the packages as stakeholders (they want the packages to be delivered in a safe, yet speedy manner), the state / governments (restricting air use in such a way to limit impact on other users) … The definition of such a mission statement represents the starting point of the design process and the requirements

definition. Furthermore, the mission statement provides the basis for the ultimate test of the system's fitness-for-purpose.

A system is therefore an assembly of system elements that interact with one another. The system is characterized by its state, its behaviour, and its external boundary. The operating environment acts on the system. These actions form an input on the systems behaviour. In the same way, the results of the systems behaviour form outputs to the operating environment world.

As mentioned, the approach chosen here is model-based. INCOSE defines Model-Based System Engineering as "the formalised application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the concept design phase and continuing throughout development and later life cycle phases" [INCOSE, 2017].

The benefits of Model-Based Systems Engineering over document-centric approaches have been acknowledged in the literature [Madni, 2018]. With an MBSE approach, a model serves as a reference for early debugging of the system under design: the earlier you detect design errors in the life cycle of the system, the more you save development costs. A model also helps preparing testing, which is one of the highest cost-prone steps in the design cycle of complex systems.

Promoters of formal methods have extensively published on simulation and verification techniques enabling early checking of models against design errors. Similarly an impressive number of papers have discussed test generation from state machines and other formal models [Dssouli, 2017]. In both cases, the widespread of proposed techniques has been hampered by the little acceptance of formal methods in industry.

In parallel to formal methods, graphic and semi formal modeling languages have emerged in research laboratories, industry and standardization bodies. With the support of tool manufacturers and system engineering practitioners, the Object Management Group and the INCOSE have jointly standardized the Systems Modeling Language (SysML) [OMG-SysML, 2017] [Friendethal, 2012]).

SysML has a wide application spectrum. Examples include trains [Baduel, 2018], helicopters [Anderson, 2010], space [Wassem, 2018], Industry 4.0 [Arantes, 2018] and clinical medicine [Khayal, 2017]. Several authors have questioned the expression power, the level of formality, and the tool support of SysML in their respective application domain.

Applying SysML to UAV needs to pay attention to the limitations the language may have in respect to real-time systems in general. The expression power of SysML is a key issue to handle parallelism, communication and time-criticality in real-time systems. Besides the syntax and the expression power of the language, its semantics is also of high importance. Having a precise and unambiguous understanding of SysML diagrams facilitates tool interoperability and sharing of models in general.

A common practice in giving diagrammatic modeling language a formal semantics is to express the semantics of the diagrams using a formal method that is mathematically defined and therefore already owns a formal semantics. The real challenge may then be phrased as follows: how to give SysML a formal semantics and to keep it user-friendly to industry practitioners who are not necessarily formal methods literate?

That question cannot be dissociated from that of the user-friendliness of the SysML tools that directly benefit from the formalisation of the language's semantics. Of prime concern are the simulation and verification tools developed for SysML. In this paper, the word "simulation" will be used to denote a possibly partial exploration of the state space of the system modeled in SysML. By contrast, formal verification relies on mathematics than chance and enables systematic exploration of the state space of the system. This is to be compared with the definitions of the IEEE 1012-2012 standard [IEEE, 2005] where "verification" guarantees the models have been correctly built and "validation" guarantees the modeled system matches the requirements.

The free and open-source SysML tool [TTool, 2018] offers simulation and formal verification capabilities. TTool includes a SysML diagram editor, a model simulator, several formal verification modules, code generators and a test sequence generator. The tool cannot be dissociated form the method of incremental modeling [Saqui-Sannes, 2016] that has already been applied to several case studies [Mattei, 2017] [Saqui-Sannes 2018]. In the paper, a novel case study - a UAV in charge of taking pictures – underlies the presentation of the entire method.

The paper is organized as follows. Section 2 overviews TTool, the SysML diagrams it supports and the method it is associated with. Section 3 specifies the UAV that serves as running example throughout the paper. Sections 4, 5, 6 and 7 respectively address the modeling assumptions elicitation, requirement capture, analysis, and design steps of the method exposed in Section 2. Discussion goes on in Section 8 and 9 to apply simulation and formal verification techniques to the diagrams presented in Section 4 to 7. At this point, a nominal system has been discussed. With limited resources instead of unlimited ones, Section 10 poses the problem of incremental modeling. Section 11 surveys related work. Section 12 concludes the paper.

2. Tool and Method

2.1. TTool

TTool is a free and open-source toolkit supporting several UML profiles, where the term "profile" denotes a variant of the Unified Modeling Language (UML) [OMG – UML, 2017] that has been tailored for an application domain. SysML/AVATAR (Automated Verification of reAl Time softwARe), hereby termed as "SysML" for simplification purposes, is one of the UML profile supported by TTool: it has been designed with systems engineering of real-time and distributed systems in mind. In brief, TTool is made up of the following main tools:

- The editor enables creation of a SysML model made up of several diagrams that individually convey a point of view on the system.
- The simulator animates SysML design diagrams and enables early debugging of SysML models.
- The model checker and the verification by abstraction module of TTool deeply explore the behaviour of the SysML model, as soon as the state space of the latter is finite.
- The test generator uses a formally verified model to generate abstract test suites.

Figure 1 uses a SysML use-case diagram to depict the main functions offered by TTool. Each use-case depicted by an oval contains one of these functions. The <<include>> relation reminds that formal verification systematically uses reachability graph construction to explore the state space of the model, as long as that state space is finite and may be explored in a reasonable time. The <<extend>> relation denotes the possibility offered to the model design not only to generate verification results from a SysML model but also to go back from the verification results to the initial model in SysML.

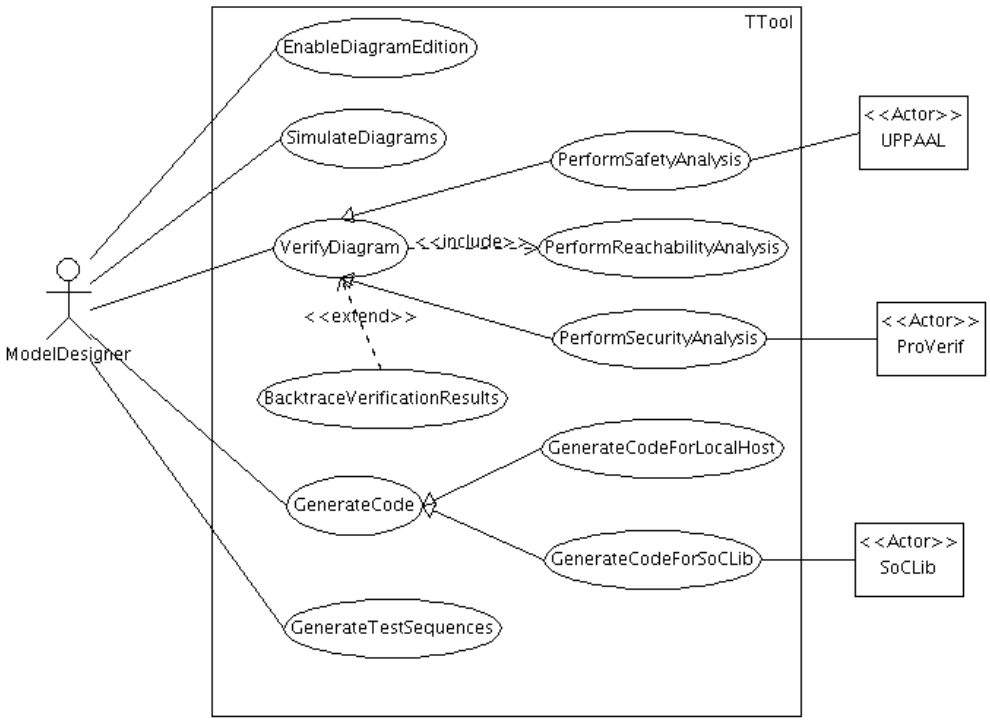

Figure 1. TTool

2.2. Method

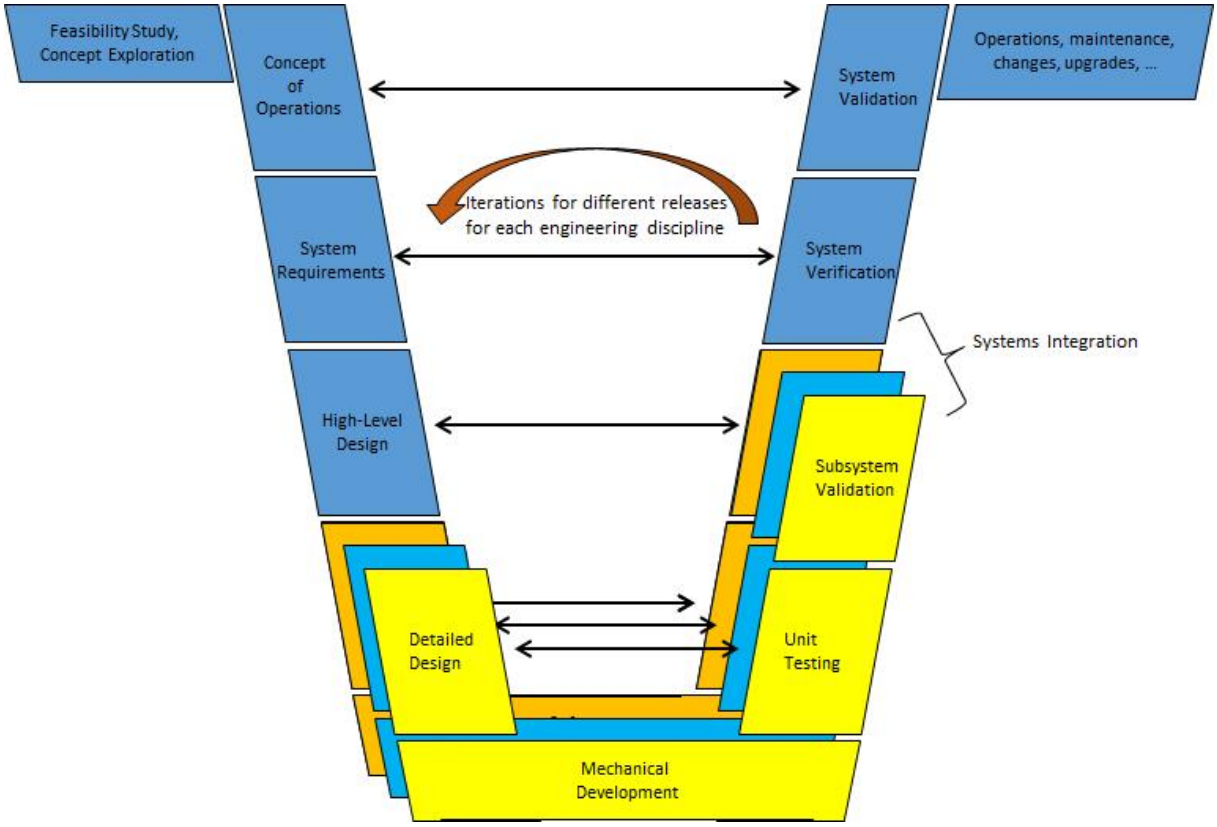

Figure 2. Typical V-cycle

Amongst the traditional development cycles, the V-cycle is the most widely cited (see Fig. 2). Starting from feasibility studies, a concept of operations, expressed user desires and needs, and several other information sources (such as a concept of operations at organisational level), a system operational concept is obtained, an expression of what the system is expected to do. As indicated by [INCOSE HandBook V4, July 2015], the system operational concept document is used to communicate overall

quantitative and qualitative systems characteristics to the acquirer, user, supplier and other organisation elements.

In successive steps, successively, the user requirements are obtained, and then, via a first architecture study, the system requirements are obtained, describing the functions the system as a whole should fulfill to satisfy the stakeholder needs and requirements. System requirements are expressed in an appropriate combination of textual statements, views, and non-functional requirements; the latter expressing the levels of safety, security, reliability, etc., that will be necessary [SEBOK, 2018].

For certain targeted functionalities, different realisation technologies can be chosen, e.g. an electronic component vs. a software implementation. These choices lead to a set of activities that can be done in each of the realisation technologies, e.g. electrical and electronics, mechanical, software. In the next steps, and in function of the chosen realisation approach for each of the retained technologies, the designs are refined step-by-step passing via higher-level designs into detailed designs. Separate development cycles take place for the detailed design phases, implementation and unit tests. Each of those development cycles has its own characteristics; for example the time that is needed for evolutions is inherently different for each of the technologies (software evolutions can be made significantly faster available for testing than for example a new layout for electronics).

At the system integration stage the complete system is being built. Verification and validation can now continue on the basis of the complete system and if final validation and acceptation by the customer take place, the preparation for production, operations, etc. can take place. An iterative approach with feedback loops allows for getting back to earlier phases to change design options, correct issues, etc., leading to an inherent recursiveness in the development process.

Model-based approaches are used to better support and guide the system designers. In the first phases of the development cycle, the Systems Modelling Language (SysML) can be used to specify the requirements and the constraints that apply on the system at hand. SysML allows for a strong inter-disciplinary cooperation. This language can be used with tool chains such as TopCased [Vernadat, 2006] and TTool [Apvrille, 2013], proposing different model checking tools to the system designer. At higher design levels these models are refined, enriched and are often linked to other models aiming at giving more insight in specific areas. Co-simulation of phenomena covering different systems, or different physical aspects of the system at hand, allows for the system designer to better understand the different impacts [Fitzgerald, 2015]. Finally, at detailed design level discipline-specific models are used for the different realisation technologies allowing for a full expression of the characteristics that are required for the specific technology.

The SysML standard at OMG defines a notation, but not a way of using it. In other words, SysML needs to be associated with a method. The one promoted in this paper applies to a large variety of real-time systems and to UAV in particular. In brief, the method is an incremental process where each increment implements a trajectory.

The trajectory (figure 3) can be sketched as follows:

- *Modeling assumptions expression*: it uses modeling assumptions diagrams to explain how the model simplifies the system.
- *Requirement capture*: it structures user and stakeholder requirements in requirement diagrams.
- *Analysis*: it is use-case driven (use-case diagram) and documented by scenarios (sequence diagrams) and flow-charts (activity diagrams).
- *Design*: it defines the architecture of the system in the form of a block instance diagram and gives each block one behaviour expressed in the form of a state machine diagram.

The first three steps of the trajectory in Figure 2 produce documentation drawing. The design step is different from previous steps since one may apply simulation and formal verification techniques to the block architecture and to the state machines modeling the inner workings of the blocks.

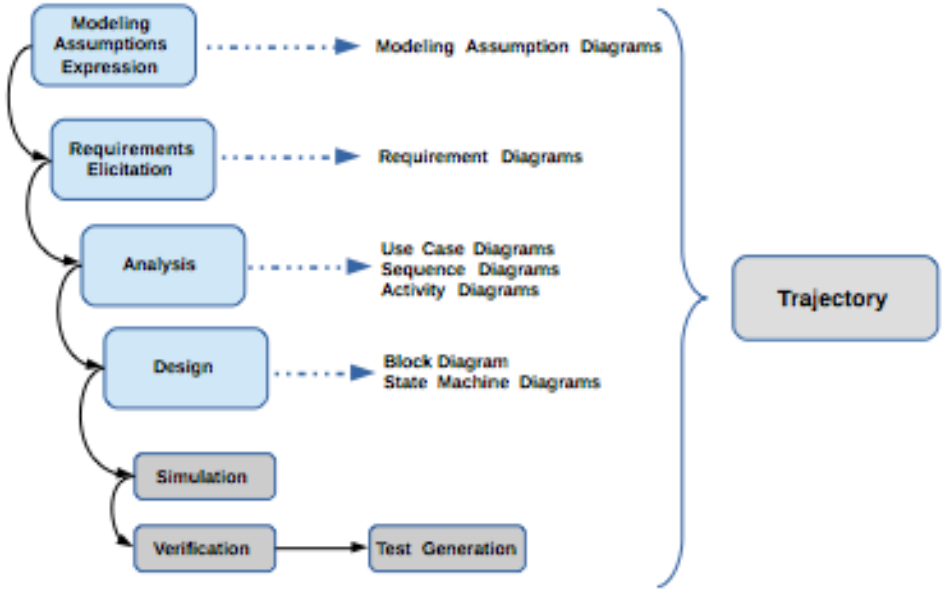

Figure 3. Trajectory and SysML diagrams

The method associated with SysML and TTool uses the trajectory in Figure 3 and shapes it in the spiral depicted by Figure 4. The spiral makes the method an incremental one. The designer is strongly encouraged to not develop a supposedly complete model of the system before starting using the simulator. Conversely he or she is advised to build a first version of the model making strong simplifying assumptions, and to progressively release the constraints associated with the assumptions each time he or she creates a new version of the SysML model of the system. This is why each trajectory in Figure 4 is associated with a new set of assumptions (depicted by an oval).

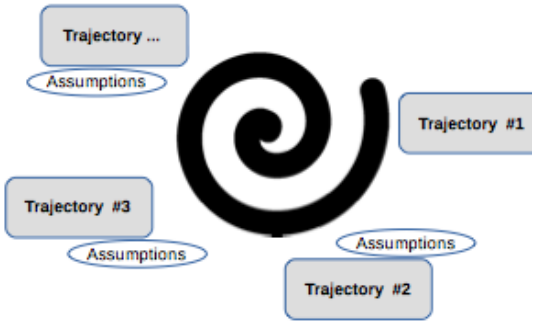

Figure 4. Incremental method

Next sections stepwise apply the method depicted by figures 2 and 3 to the UAV that serves as running example throughout the paper.

3. Case Study

The UAV can autonomously take off, fly in a stabilized way, and land at its destination or whenever a critical situation is encountered. It takes pictures at given locations. Only the software related to taking pictures is modeled in this case study: the taking off, flying and landing actions are not modeled.

Pictures can be taken only when the drone is flying. A remote system located in a ground station can send picture order to the drone. A picture order contains the GPS position of the picture to be taken. To know its current position, a drone has an integrated GPS. When a picture GPS point is reached, with regards to a given threshold, the picture is taken, and then stored on a CompactFlash removable storage system. The system needs 2 seconds to take a picture, and between 4 and 5 seconds to store it in on the memory card. Pictures may be remotely downloaded from the ground station using a download order. Pictures can also be read from the CompactFlash once the drone has come back from its mission.

4. Modeling Assumptions Expression

4.1. Rationale

Experience has shown that models are scarcely self-contained and need to be documented to facilitate their sharing and reuse. In particular, a model remains hard to understand for somebody who does not know about the simplifications and more generally the assumptions made by the model's designer. Therefore, the authors of the paper advocate for an explicit inclusion of modeling assumptions inside the model of the system. The Modeling Assumption Diagram, which is not part of the SysML standard [OMG-SysML 2017], has accordingly been introduced into the version of SysML supported by TTool [Saqui-Sannes, 2016].

4.2. Restricting the model to its core behaviour

The life of a system is not limited to the core behaviour of that system. To become active, a real system usually needs to go through an initialization procedure. Similarly, a system needs a shutdown procedure to be definitely interrupted or temporarily interrupted for maintenance. Surprisingly, the SysML models usually presented in the literature do not address the initialization and shutdown procedures. Nor they address maintenance.

For the sake of simplicity, this paper also ignores the initialization, shutdown and maintenance procedure associated with the UAV. But this important simplification is made explicit model by using a Modeling Assumption Diagram (Figure 4) included into the SysML model of the UAV.

In terms of syntax, a MAD defines a tree-structure of "boxes" that contain the modeling assumptions. Pairs of "boxes" may be linked. In Figure 4, the containment depicted by a cross inside a circle allows one to split up a complex assumption into elementary ones.

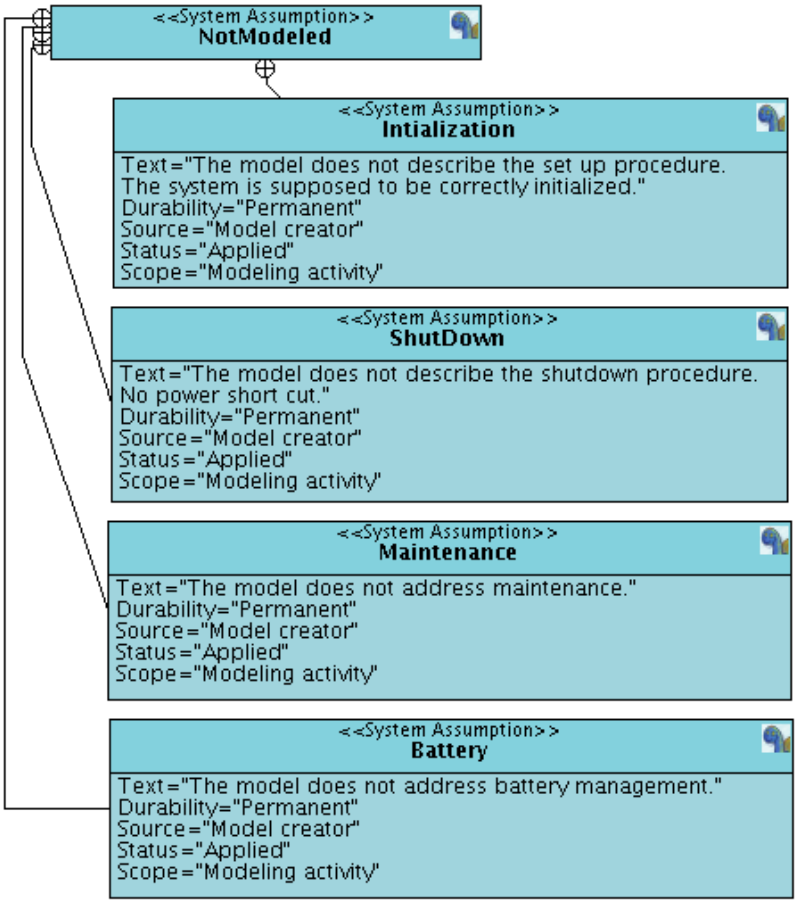

Figure 4. Making explicit what the model ignores

4.3. Use of resources

It is commonplace to write the resources a system can access are necessarily limited. It is less common to see SysML models clearly enumerating to what extent each resource accessed by the system is addressed by the SysML model or not.

MADs may help changing that situation. In figure 4, the assumption linked to the battery says the latter is not managed. The *Durability* attribute being set to "Permanent", the reader of the model does not expect any forthcoming version of the model to manage the battery.

4.4. Versioning

Again, the method associated with SysML and TTool is incremental. What one may expect from a MAD is an assistance to manage versioning. Figure 5 exemplifies it in the context of a UAV model of which two versions do exist in the same SysML model. First version of the model assumes the UAV takes one picture whereas version 2 allows the ground station to parameter the number of pictures taken by the UAV. Also, first version of the model considers the compact flash memory has an unlimited capacity whereas version 2 of the model increases the memory capacity up to three pictures. Finally, versions 1 and 2 of the UAV model differ by the absence and presence of turbulence, respectively.

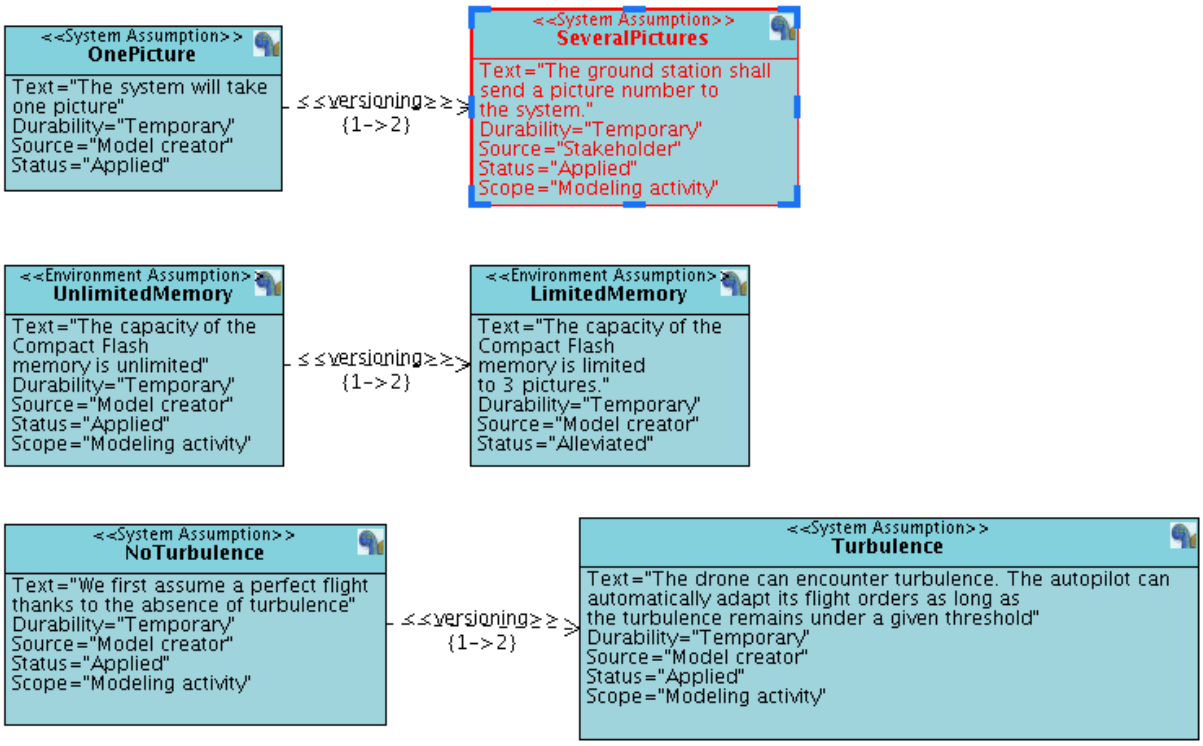

Figure 5. Versioning

Modeling assumptions usefully complement the requirements captured from users and stakeholders. The role of assumptions nevertheless remains different from the one played by the requirements and therefore SysML separately expresses the requirements in requirement diagrams.

5. Requirements

5.1. Requirement diagram

The goal of system architecture activities is to define a complete solution based on principles, concepts and properties logically related and consistent with each other. Such solution should have suitable characteristics and properties, matching as well as possible to the problem expressed by a set of system requirements,

traceable to mission/business and stakeholder requirements, and traceable throughout life cycle phases and corresponding engineering tools (e.g., mechanical, electronics, software). This underlines the necessity to obtain pertinent requirements and explains why SysML supports requirement diagrams, a type of diagram not taken on board by UML.

As SysML is a language and not a method, there are no constraints to the writing style of requirements. In contrast, an advantage of requirement diagrams is to oblige the system designer to structure and organize the requirements, and to show how the latter relate to other diagrams in the model.

5.2. Writing good requirements

Good requirements definitions are vital to successfully design and develop systems. It is the role of the systems engineer to elicit the desires and the needs from users and stakeholders to ensure the product will meet their needs. He/she then reformulates those desires and needs into requirements in such a way that they can be successfully used as input for the development process [NASA Systems Engineering Handbook].

Several points are important for "good requirements" [NASA Systems Engineering Handbook]. First of all, it is important to use active phrases as opposed to passive voice, e.g. "the brake response time shall be inferior to 5ms". "Shall" is to be used for requirements, whereas "Should" is to be used for goals". Then, it is important to use consistent terminology throughout the requirements for the system and its sub-systems. The phrases have to be grammatically correct, free of typos, misspellings, and punctuation errors, so to facilitate unambiguous understanding and interpretation of the requirements.

Without aim of exhaustiveness, the set of requirements should be

- Good and clearly defined – understandable in an unambiguous manner
    - Example: "The UAV shall carry parcels of up to 25 kg over a minimum distance of 20 km"
- Complete with well-defined associated assumptions…
    - Example: "Given the air-infrastructure limitations, the UAV shall not fly over an altitude of 100m above ground and shall not fly faster than 20km/h ."
- Implementation-free definition at the correct level (i.e., system, element, subsystem)…
    - Example: "The UAV flight control system shall allow positioning of the UAV at a precision of ±10cm and a speed accuracy of ±1m/s"
- Consistent amongst the system's requirements and with requirements of related systems, using consistent terminology, …
    - Example: "The parcel size the UAV will need to carry shall not exceed 40cm x 40cm x 40cm" (no basic contradiction with the weight requirement expressed above)
- Traceable to higher-level requirements – are all requirements necessary?
    - Example: "The GPS precision necessary for the positioning shall be less than 5cm"
- Correct and technically feasible
- Verifiable/Testable – Can the system be tested, demonstrated, inspected, or analysed to verify adherence to the requirements and are requirements stated precise enough to facilitate specification of system test success criteria and requirements?

Looking at functional, performance and integration, again without exhaustiveness, the following aspects need to be taken into account:

- Functional – Described functions necessary and sufficient to meet the system needs, goals, and objectives
- Performance – Required performance specified and margins listed
    - Example: "The UAV will have a maximum speed of 20km/h"
- Interfaces – All external and internal interfaces clearly defined
    - Example: "The UAV receives flight instructions via radio signals, specific band to be defined"

These characteristics of "good requirements" do not only apply to SysML, they apply for all possible requirements specification methods.

Each node depicted by a box contains one requirement together with its unique identifier, a text, and a categorization between functional and non-functional requirement.

5.3. Relations inside a requirement diagram

Usual tabular representations of requirements merely list the latter and possibly organize them into chapters. Conversely, a SysML requirement diagram invites the model creator to organize the requirements in a more structured way. It indeed organizes requirements in a tree structure where pairs of requirements nodes are connected by either of the following relations:

- *Containment*. An arrow terminating by a cross and surrounded by a circle allows one to split up a high level requirement into elementary ones.
- *Refinement*. The <<refine>> relation from R1 to R2 allows R2 to add more precision to R1.
- *Derivation*. The <<deriveReqt>> relation from R1 to R2 allows R2 to bring a technical solution to functional requirement R1.

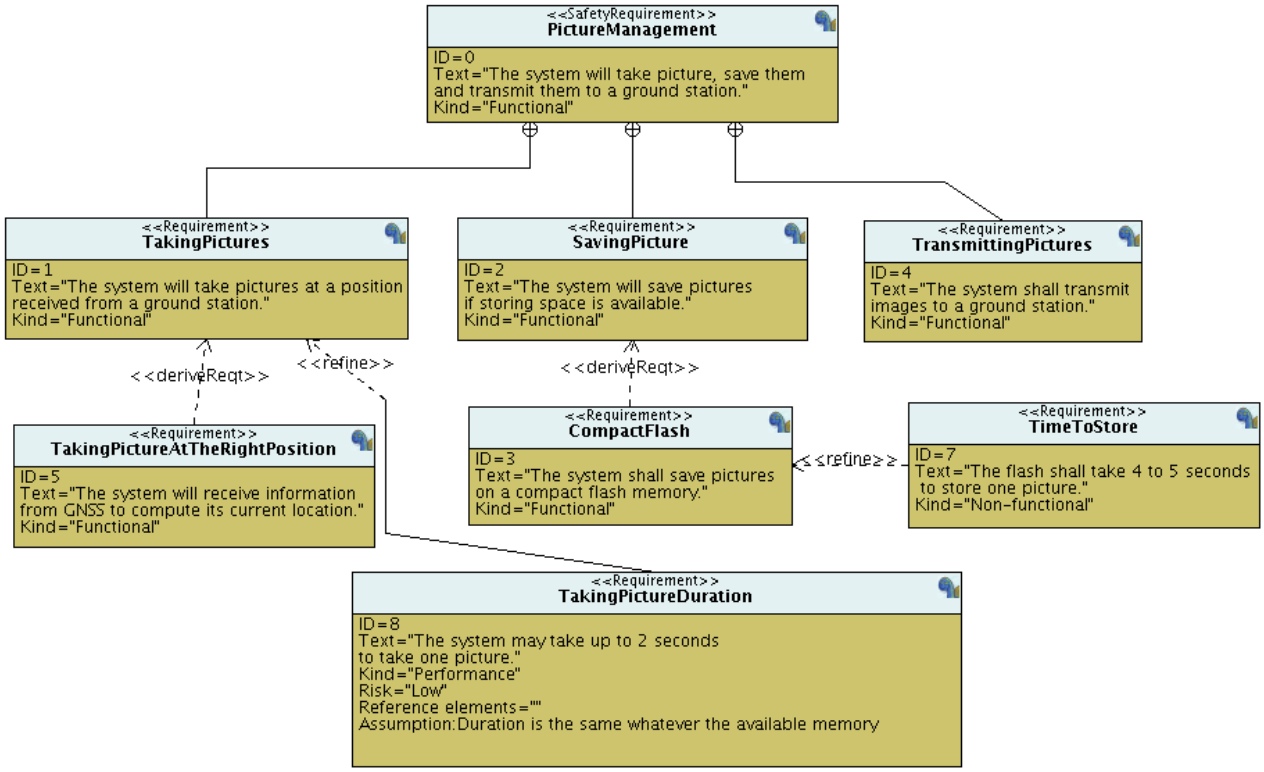

Figure 6. Requirement Diagram

5.4. Relations between the requirements and other diagrams of the same model

Figure 7 extracts requirements from Figure 6 and uses the <<satisfy>> relation to link these requirements to other diagrams belonging to the same SysML model. *TakingPicture* is linked to the sequence diagram (Figure 9) developed during the analysis step of the trajectory introduced in Section 2. *TakingPicture* is linked to a state machine diagram developed during the design step of the method depicted by Figure 9.

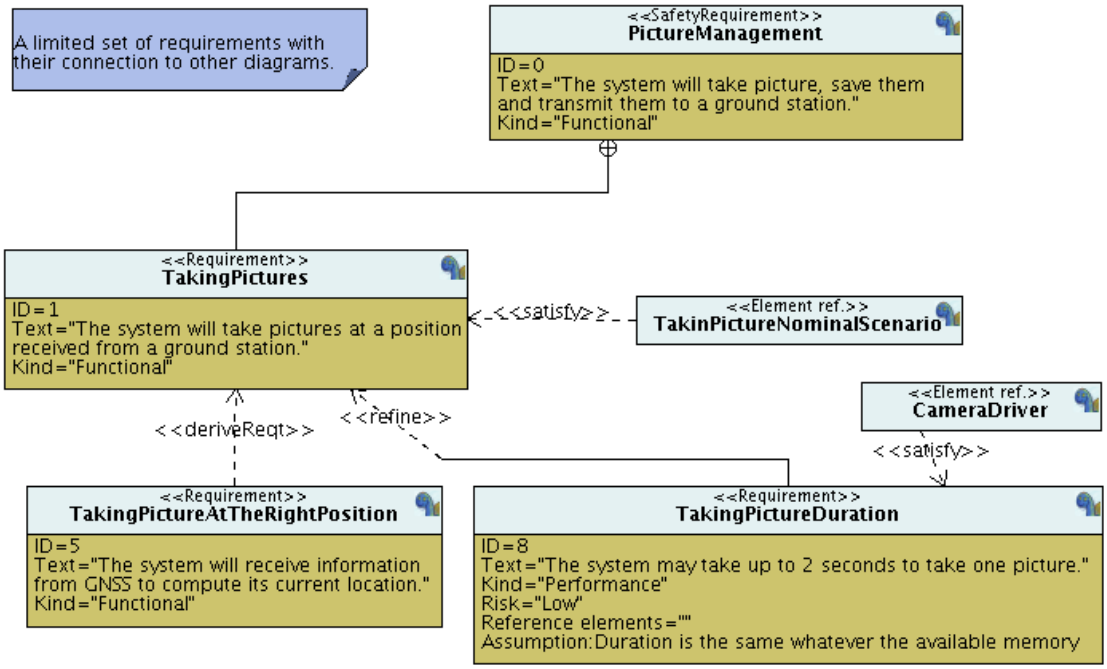

Figure 7. Requirements Linked to Other Diagrams

Whether the SysML standard [OMG-SysML, 2017] introduces a diagram specifically dedicated to requirements, the requirement elicitation process may be pursued during the analysis step.

6. Analysis

6.1. Delimiting the perimeter of the system

Creating a SysML diagram is a matter of decision-making. One of the major decisions the model designer has to take is to clearly delimit the perimeter of the system he or she intends to design and possibly to develop. The use-case diagram, a pillar of what is termed as “use-case driven analysis”, brings solution to assist the model designer in this task.

The first action the use-case diagram creator has to take is to draw a rectangle characterizing the boundary of the system. What the designer puts inside the rectangle is what he or she promises to design and develop. What he or she places outside the rectangle refers to the environment of the system and, clearly, designing the environment is not part of the duties of the SysML model creator.

In terms of SyML syntax, the rectangle contains a set of “rugby balloons” that materialize the uses-cases containing the high-level functions and services to be offered by the system. Part of these functions makes the system interact with its surrounding environment and the existence of these interactions is depicted by links connecting the use-cases to so-called “actors” belonging to the environment. Other types of links express relations between pairs of use-cases located inside the rectangle that delimits the boundary of the system.

6.2. Writing good use-cases

Again, the use-case diagram identifies the main functions and services offered by the system. A common mistake is to create a use-case that is not a high-level function, but an elementary one. If one considers a drink machine controller, a correct use-case may be *Process Payment*. Actions such as *Compute Money* or *Return Change Back* are elementary actions one should not be developed in a use case diagram but within an activity diagram documenting a use case.

On the other hand, the name of the use-case, usually driven by a verb, must convey the point of view of the system, not the point of view of the actors. Again, for a drink machine controller, *Process Payment* conveys the point of view of the system (the controller) whereas *Insert Coin* would convey the point of view of the user of the coffee machine.

The rules established in this section have been taken into account to develop the use-case diagram depicted by Figure 8.

6.3. Relations between use-cases

Figure 7 depicts the use-case diagram developed for the UAV model. The diagram uses <<include>> relations between pairs to denote function inclusions: for instance, *ManagePicture* necessarily includes two auxiliary functions to take and send pictures, respectively.

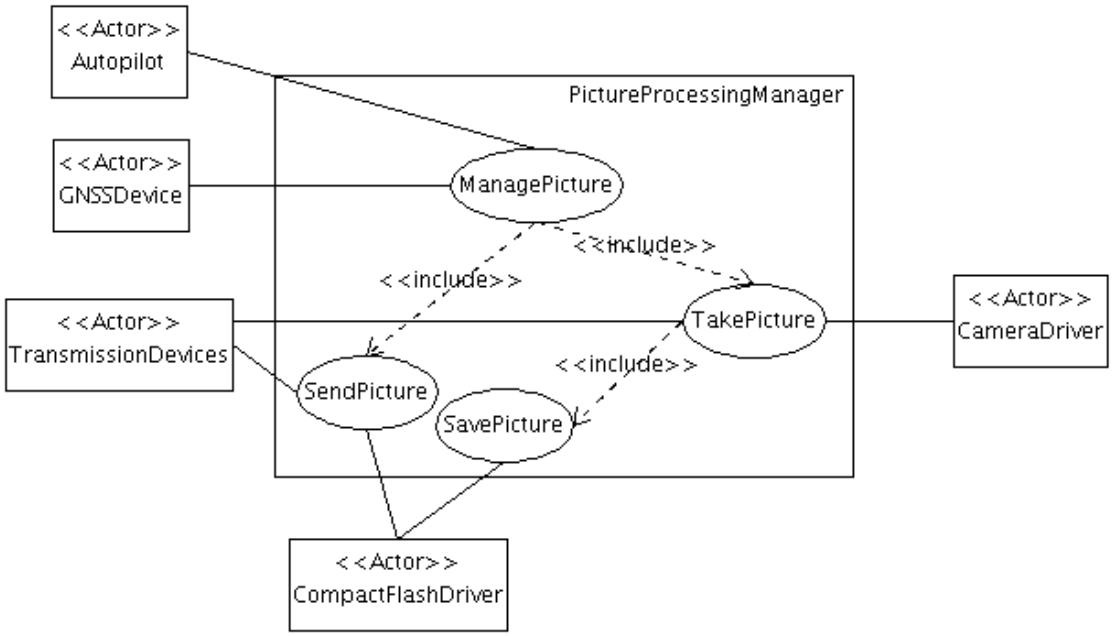

Figure 8. Use-Case Diagram

A use-case diagram shows how the main functions of the system interact with the outside environment. Nevertheless the use-case diagram does not specify the type of messages or signals the system uses to interact with the actors belonging to its environment. That role is dedicated to the sequence diagrams that we use to document use-case in the form of scenarios.

6.4. Documenting use-cases

Figure 9 depicts the scenario developed for the UAV model and more precisely for the first version of that model. Consequently Figure 9 depicts a nominal scenario for a perfect system and a perfect environment. At this point, the UAV is not supposed to run out of battery. Nor its compact flash memory can be saturated. The sequence diagram shows how the UAV receives an order from the ground station, check its current position against and take corrective actions, saves the pictures and transmit it to the ground station. Conforming to the assumption diagram depicted by Figure 5, the first version of the UAV takes one picture.

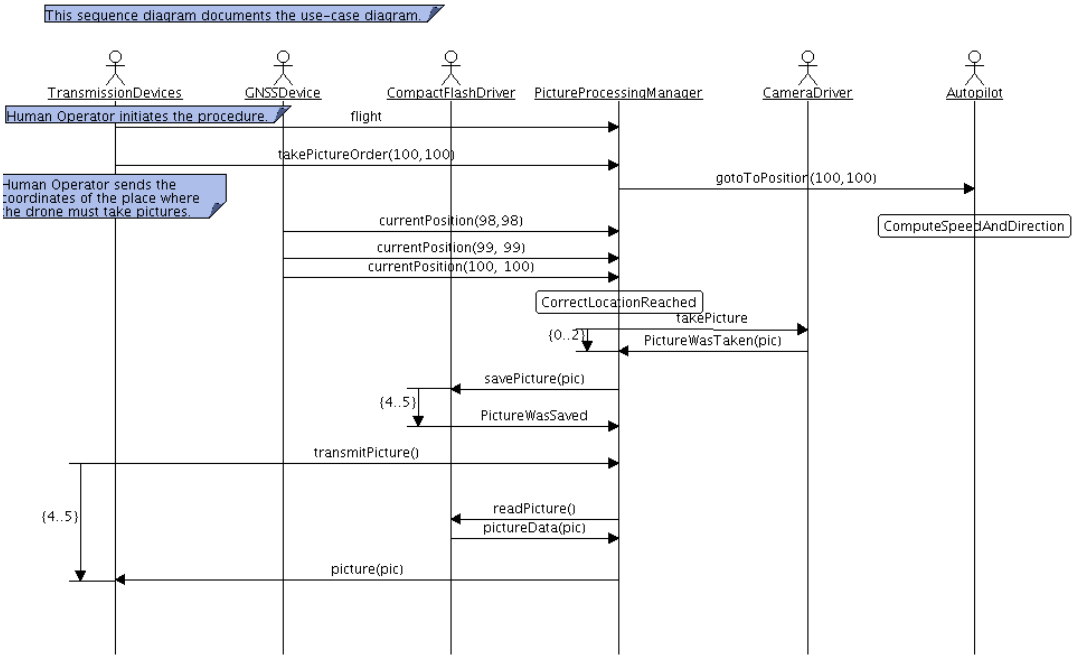

Figure 9. Sequence Diagram modeling a nominal scenario

A sequence diagram represents one possible execution scenario. In no way it is intended to represent the entire behaviour of the system. Nor it lists all the messages exchanged between the systems and its environment, or the messages exchanged inside the system. The complete list of exchanged messages will appear on the architecture modeled during next step, namely the design step. Nevertheless, developing one or several sequence diagrams helps the model designer bridging the gap between the use-case diagram and architectural design, given the former and the latter are functionally- and object-oriented, respectively.

7. Design

The design steps first addresses a fundamental issue: elaborating and fixing the architecture of the system (section 7.1). In SysML, the architecture remains a static structure depicting the system as a set of interconnected "boxes." Adding state-machines diagrams (Section 7.2) to the model enables describing the inner workings of the architecture blocks and makes the model an executable one that may be processed later on by simulation and verification pieces of software.

7.1. Architectural Design

With SysML, the system engineering community has developed its own modeling language to depart from the software engineering community. At the same time, the choice was made to rely SysML on UML. Therefore, SyML *de facto* reuses the object-oriented principles of UML even at the price of renaming "class" by "block" and describing system architecture by a block diagram in lieu of a class diagram. Therefore, elaborating a system architecture boils down to not a "finding objects" problem but to a "block finding" one.

As far as real-time and distributed systems are concerned, the system architecture is primarily a set of interconnected blocks that exchange signals. Elaborating a SysML block diagram therefore requires one's capacity to enumerate the entire list of exchanged messages and to assign them to relevant blocks as input or output signals. As soon as the interfaces are clearly defined in terms of signals, the model designer may add attributes to the blocks as premises to defining the state machines in charge of sending and receiving the signals (behavioural design modeling is the subject of next subsection).

The standardized version of SysML uses two architectural diagrams: the block diagram and the internal block diagram. By contrast, the version of SysML supported by TTool supports one architectural diagram: the block instance diagram, an example of which is depicted by Figure 10 to structure the UAV that serves as running example.

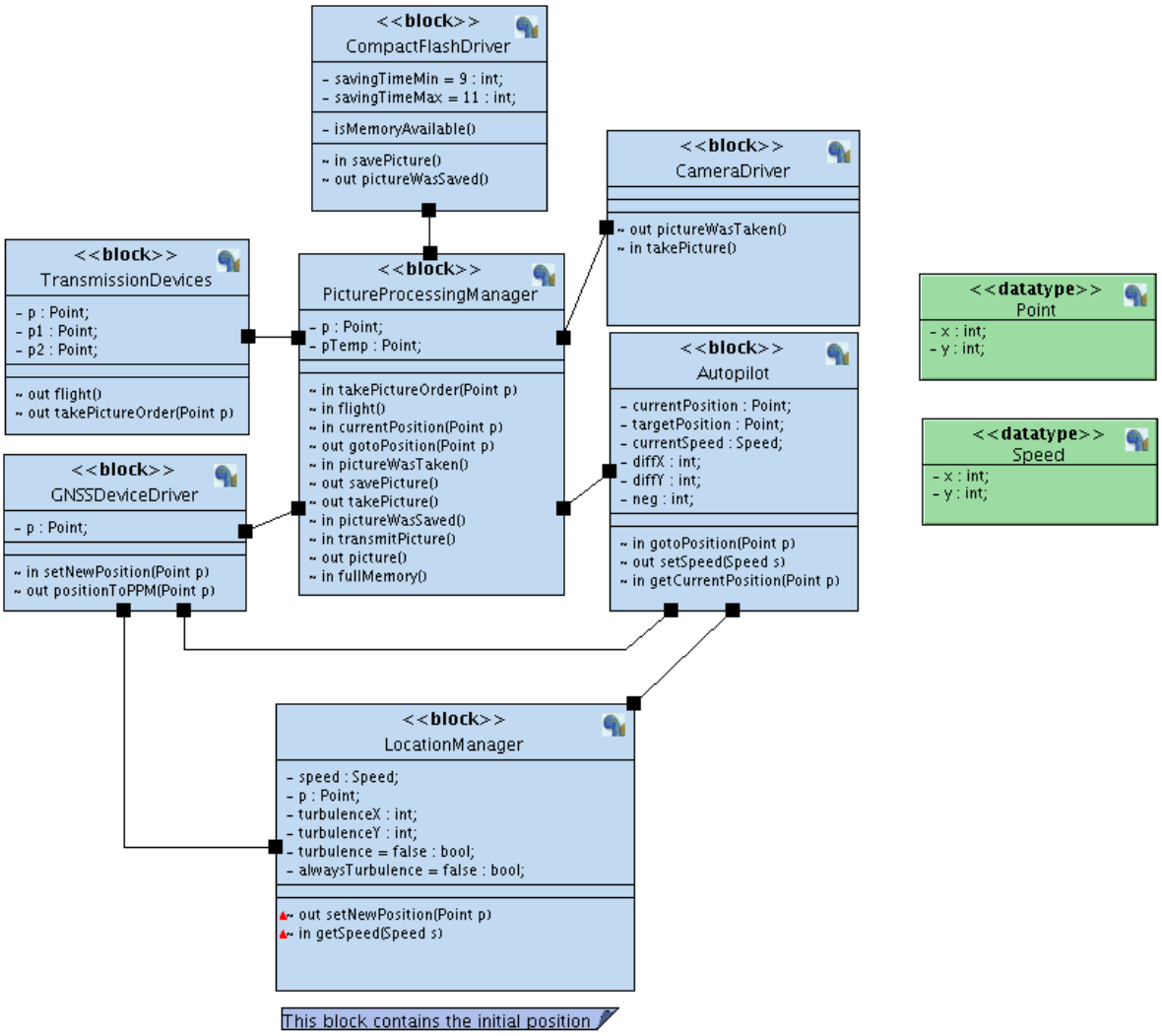

Figure 10. Block Diagram

As soon as the architecture is stable in terms of exchanged messages, one may start defining the inner workings of the blocks in the form of state machines.

7.2. Behavioural Design

Various paradigms might be used to describe the behaviour of the blocks a SysML architecture is made up of. Unlike SCADE [Lesergent, 2011], which mixes flow-control and state machine operators to model real-time software, SysML supports one behavioural description paradigm: the state/transition description style expressed in state machine diagrams. Clearly, this paradigm advantages the type of systems where the control part precedes the data part and this perfectly fits in with type of the real-time systems discussed in this paper.

For readability reasons, this section does not present the complete set of state machines developed for the UAV model. The state machine in Figure 10 is one of the simplest state machine in the model, but it suffices to exemplify the type of extended communicating finite state machine supported by TTool: states machines that handle variables (block attributes), message reception/emission, and time (delay on transitions).

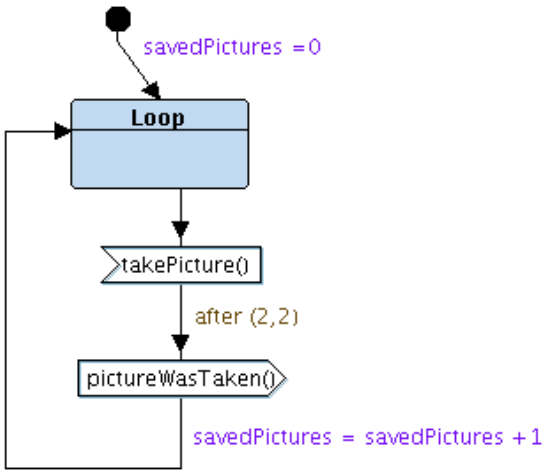

Figure 11. State Machine Diagram for the Memory Recorder

The state machine depicted by Figure 10 gives the Memory Recorder one behaviour and makes it executable, a premise to applying model simulation techniques.

8. Simulation

8.1. Principle

The Simulator of TTool (Figure 12) enables animation of state machine diagrams. It takes as input a syntactically- and type-checked SysML model and computes the model's initial global state. Step by step firing of transitions enables early debugging of the model by joint observation of simulation traces in the form of sequence diagrams, annotations on the SysML model itself and display of the state, variables and other elements contained in the blocks the system is made up of. Random firing of transitions enables further exploration of the system's behaviour until a deadlock situation or a termination state is encountered.

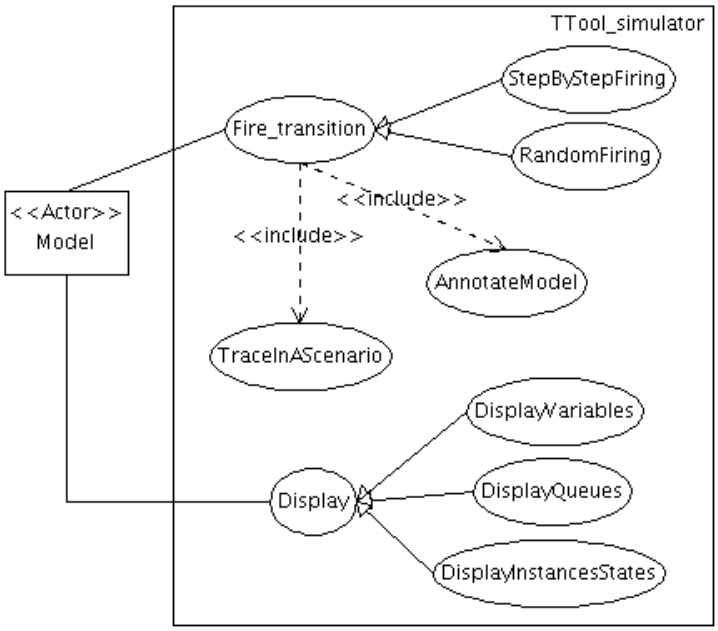

Figure 12. Simulation Capabilities of TTool

The screenshot in Figure 12 shows the first steps of the UAV model simulation. TTool outputs the simulation trace in the form of a sequence diagram that can be manually compared to the sequence diagrams and to other diagrams elaborated in the previous steps of the trajectory.

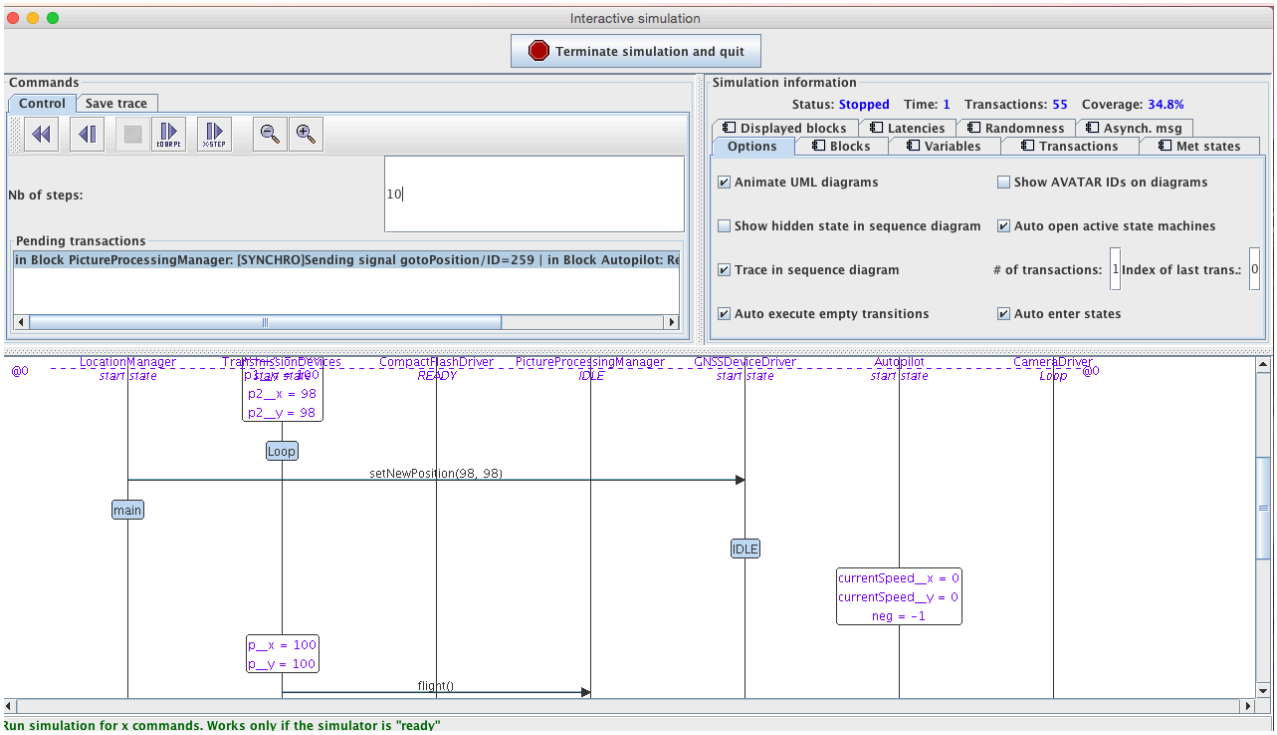

Figure 13. Simulation Trace

The simulator of TTool also animates the state machines. For example, Figure 14 depicts an animated version of the state machine introduced by Figure 11. One may understand which portion of such and such state machine has been visited and sometimes understand why the simulation blocks although not all state machines have been utterly visited.

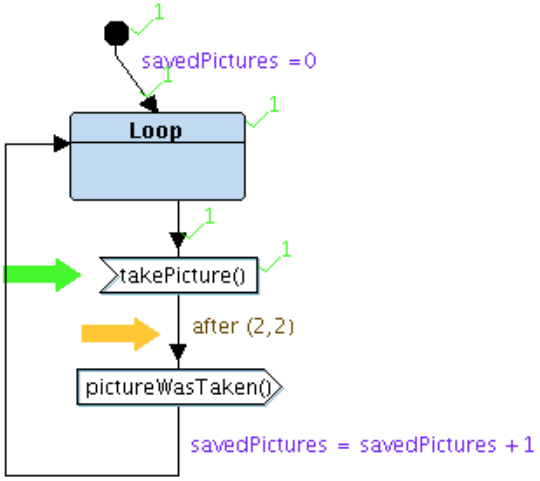

Figure 14. Following up Simulation on a State Machine

The simulation of TTool is thus of great help to explore particular execution paths in the state space of the global system. But simulation heavily depends on the expertise of the model creator in terms of capacity to pick the most relevant exploration paths. Formal verification, which is the subject of next section, is a much more systematic approach.

9. Formal Verification

9.1. Principles

One of the most widespread verification approaches is reachability analysis. Relying on a systematic analysis of the state space of the system under design, reachability analysis outputs a so-called "reachability graph" representing all the valid execution paths and states of the system starting from its initial state.

Reachability analysis faces the well known "state explosion problem" when the reachability graph of the model cannot be computed in reasonable time or not computed at all. In the remainder of the paper, we assume the reachability graph can be computed in acceptable time and manageable use of computer resources such as memory.

The reachability graph being computed, a question arises: how to exploit it to verify the desired properties of the system? Two families of answers are considered in this paper (Figure 15): model checking and verification by abstraction:

- *Model checking*: the tool checks whether one property is met or not, and outputs a yes/no answer.
- *Verification by abstraction*: the tool computes the reachability graph of the SysML model as a Labelled Transition Systems and applies minimization techniques to obtain an abstract view of the system.

The use-case diagram in Figure 15 identifies the main function involved in the verification process describes by subsequent sections. Figure 16 conveys another point of view on the same subject and exemplifies the outputs of the verification process main functions.

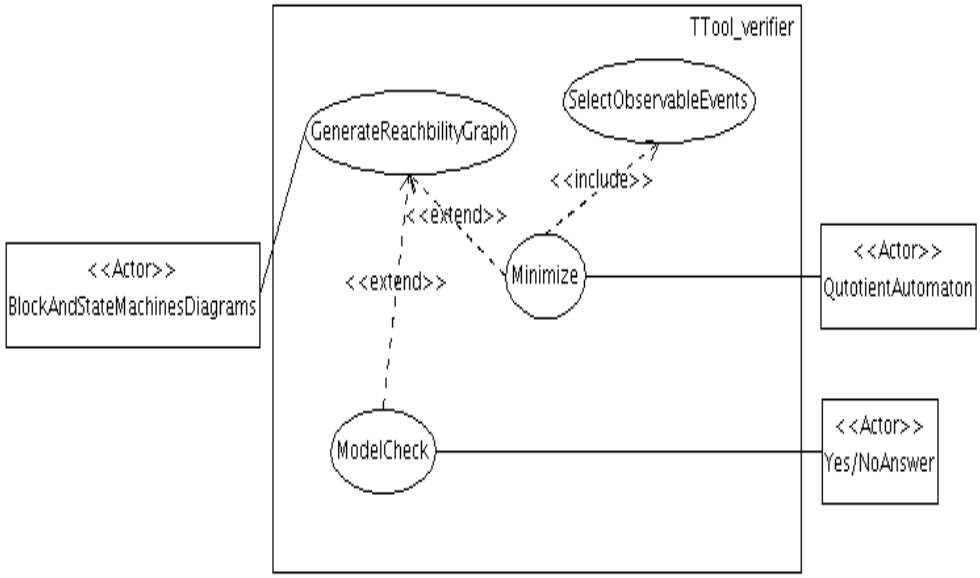

Figure 15. Verification Capabilities of TTool

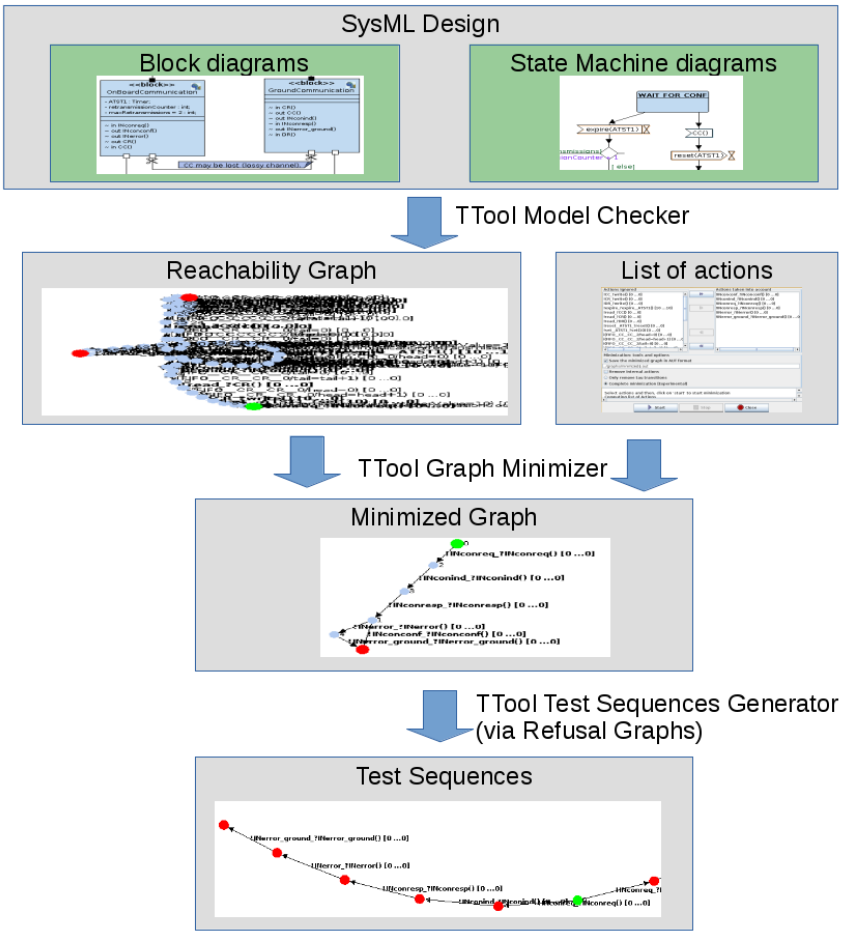

Figure 16. Formal verification and test generation

Figure 15 depicts a link between formal verification and test sequence generation. Testing is goes beyond the scope of this paper, which focuses on reachability analysis and the way one may exploit the reachability graph.

9.2. Reachability analysis of the UAV model

Figure 18 depicts the reachability graph of the first version of the UAV model. It accounts 79 states and 103 transitions. The red disk materializes a sink state, i.e. a state no transition departs from.

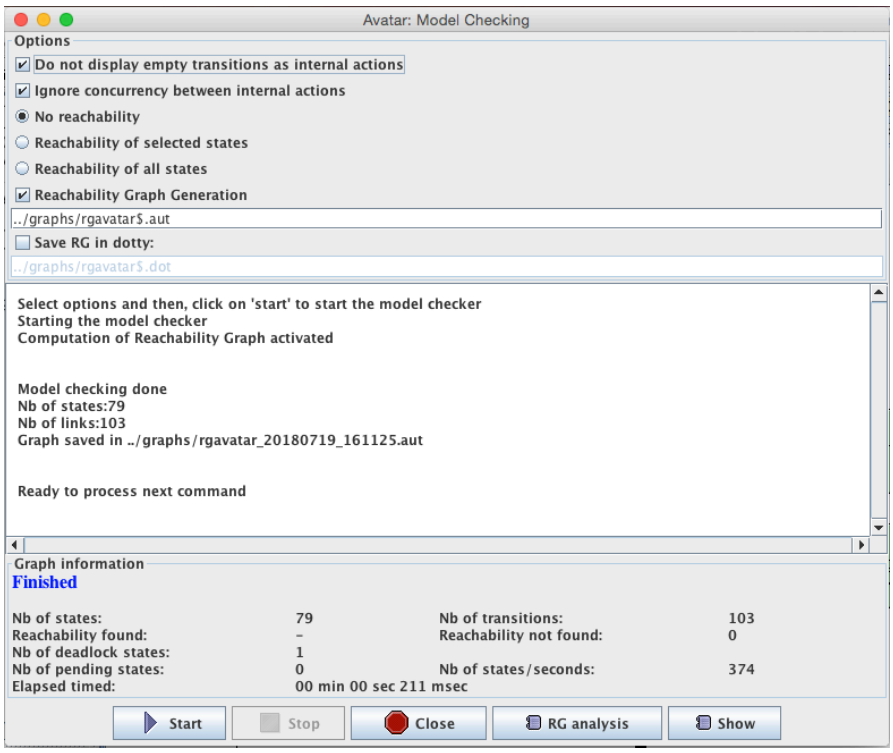

Figure 17. Reachability Analysis Interface

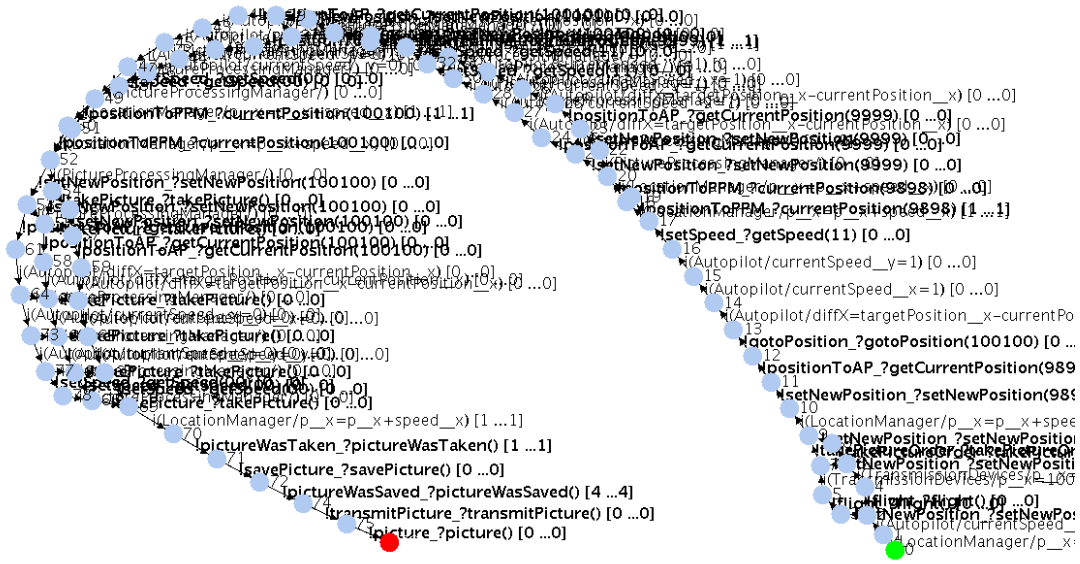

Figure 18. Reachability Graph of the UAV Model

Obviously, the reachability graph cannot be easily explored without the assistance of tools. Applying model checking techniques is a first option to answer that need.

9.3. Model Checking with the native Model Checker of TTool

Figure 19 sketches the main steps of a model checking activity.

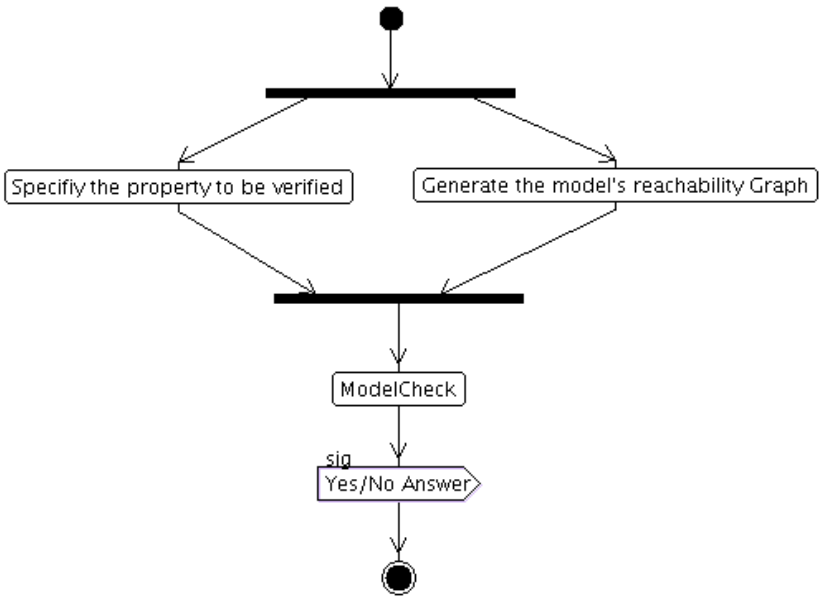

Figure 19. Principles of Model Checking

With its integrated model checker, TTool departs from SysML tools (e.g. [Rhapsody, 2017]) that use an external model checker. No need to express properties in terms of logic formulas. To check whether a state in a state machine can be effectively accessed during the execution of the model, one labels that state with RL?, an abbreviation for "Reachability Liveness ?".

Reachability and liveness are two families of properties that can be defined by the following sentences:

- *Reachability*: "Nothing bad will ever happen."
- *Liveness*: "Something good will eventually happen."

Figure 21 depicts a state machine annotated by "RL?" labels. The screenshot in Figure 22 shows that all annotated states are reachable.

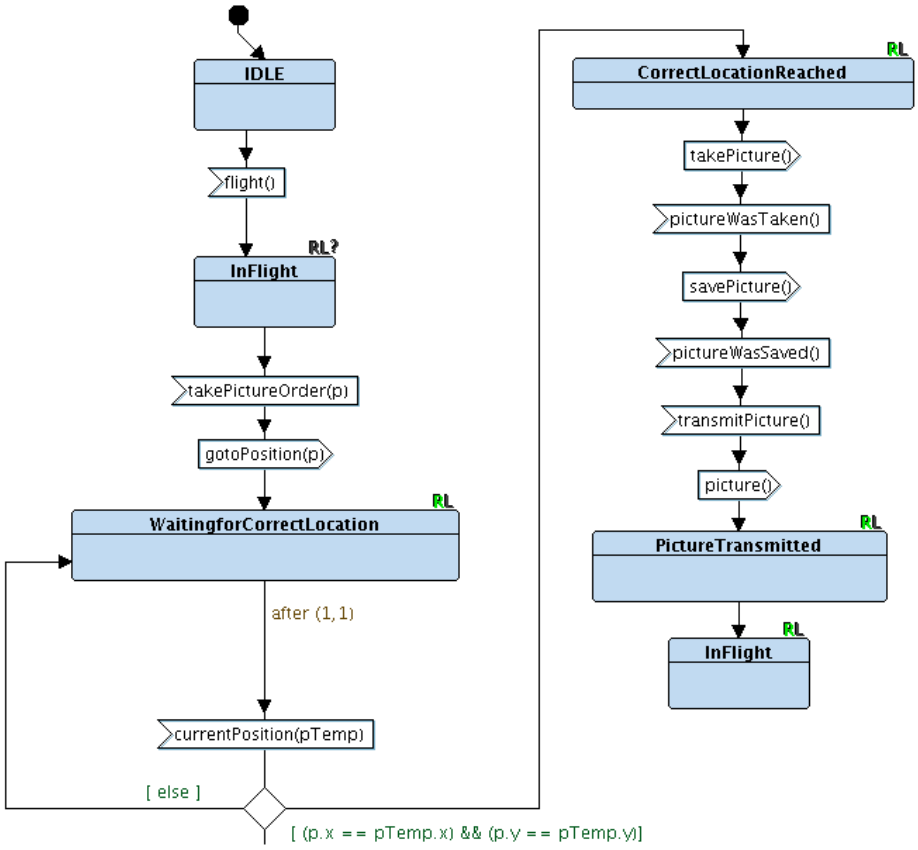

Figure 21. State Machine Diagram Annotated to Check Reachability of States

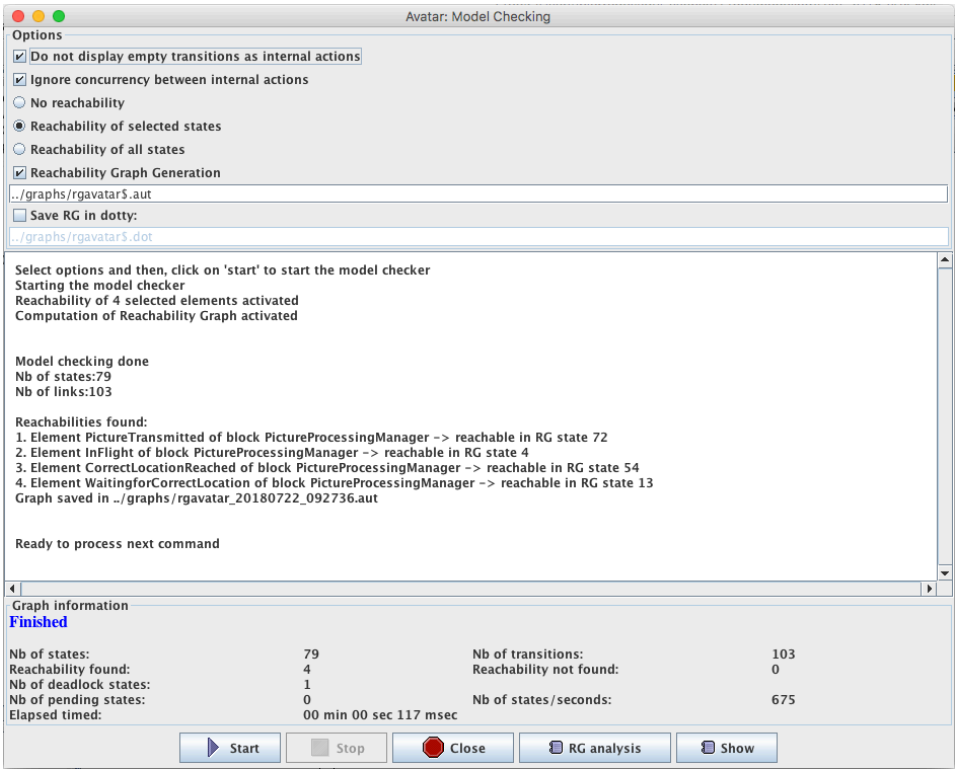

Figure 22. Model-checking Interface

The model checker integrated to TTool can be used without writing a piece of formal code and without expressing properties in terms of formulas. This is fully convenient when the property ones desires to verify can be expressed in terms of state or action reachability. For other type of properties, TTool offers an interface to model-checkers [UPPAAL, 2018].

9.5. Model Checking with Temporal Logic Formula

Safety pragmas can be used in design models in order to capture complex properties expressed in a reduced form of CTL. After checking the syntax of these pragmas, TTool can automatically invoke UPPAAL [UPPAAL, 2018] in order to verify these pragmas.

Pragmas must follow the following format:

- A[] p means that whatever the state of the modelled system, p must be satisfied.
- A<> p means that p must be satisfied in at least one state of all possible execution paths.
- E[]p means that p must be satisfied in all the states of at least one execution path.
- E<>p means that p must be satisfied in at least one state of one execution path.
- p->q means that whenever p is satisfied in an execution path, q will eventually be satisfied in the same execution path.

At the SysML model level, the logic formulas complying with the above syntax are included into the block instance diagram depicting the architecture of the system. Figure 23 shows not the entire block instance diagram, but a part of it that lists the logic formula expressing the properties to be verified by UPPAAL. Figure 24 shows the result of the verification process.

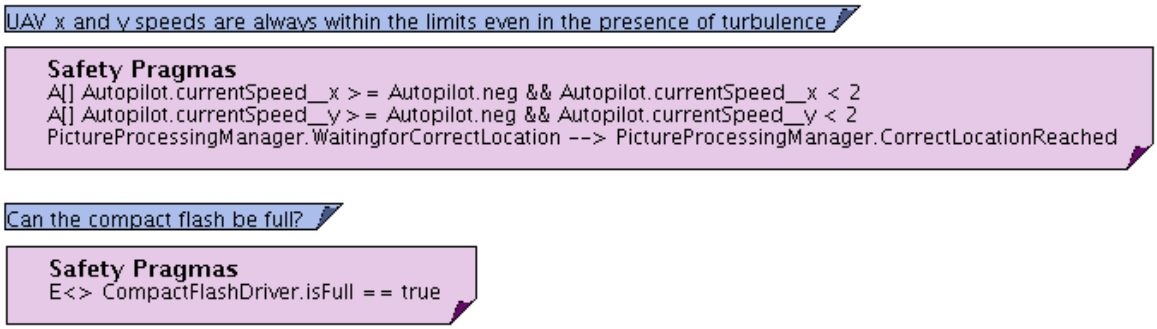

Figure 23. Safety Pragmas

```
Select options and then, click on 'start' to start generation of RG
Session id on launcher=1
Sending UPPAAL specification data


-----------------------------------------
Studying custom CTL formulae
A[] Autopilot.currentSpeed__x >= Autopilot.neg && Autopilot.currentSpeed__x < 2
A[] Autopilot__5.currentSpeed__x >= Autopilot__5.neg && Autopilot__5.currentSpeed__x < 2 -> property is satisfied
A[] Autopilot.currentSpeed__y >= Autopilot.neg && Autopilot.currentSpeed__y < 2
A[] Autopilot__5.currentSpeed__y >= Autopilot__5.neg && Autopilot__5.currentSpeed__y < 2 -> property is satisfied
PictureProcessingManager.WaitingforCorrectLocation --> PictureProcessingManager.CorrectLocationReached
PictureProcessingManager__3.id80--> PictureProcessingManager__3.id88-> property is satisfied
E<> CompactFlashDriver.isFull == true
E<> CompactFlashDriver__2.isFull == true -> property is satisfied

All Done
```

Figure 24. Results of Model-Checking with UPPAAL

9.6. Verification by Abstraction

Like model checking, the verification by abstraction approach discussed in this section has a formal background. It reuses techniques developed for process algebra [Milner, 1980] and was already applied to modeling techniques such as Petri nets [Courtiat, 1984] and Estelle [Courtiat, 1992]. The approach is not new but its application in the context of SysML is specific to TTool.

Verification by abstraction can be sketched as follows. First, the reachability graph of the SysML model is computed from the block and state machine diagrams elaborated during the design step. Second, the user of TTool reviews the list of signals exchanged by connected pairs of blocks and retain those of

interest for checking the system against its expected properties. For instance, the designer of the UAV model may decide to limit the view of the system to receiving instructions from the ground station, taking one picture, saving it on memory and transmitting it to the ground station.

The signals of interest being selected, the reachability is computed in the form of a Labeled Transition System. The reachability graph transitions are labeled in the following way: each transition that sends or receives one of the signals selected by the user of TTool is labelled by the name of the signal. Other transitions not concerned by exchanging one of user-selected signals are systematically labelled by a "nil" symbol.

The thus labelled reachability graph is the minimized in order to remove, as much as possible, the "nil" symbols and to preserve all the transitions involving an exchange of signals selected by the model creator. The minimization process uses Milner's observational equivalence [Milner, 1980] and outputs a so-called "quotient automaton".

Figure 25 sketches the verification process in a more detailed manner. Figure 26 presents the interface provided by TTool to select observable events. Figure 27 depicts the quotient automaton obtained by labelling the reachability graph depicted in Figure 18.

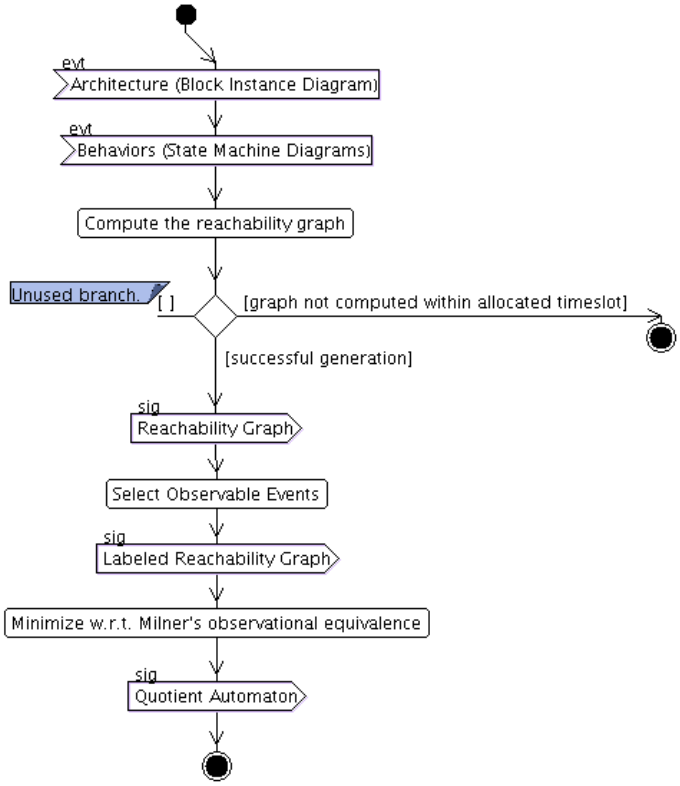

Figure 25. Verification By Abstraction

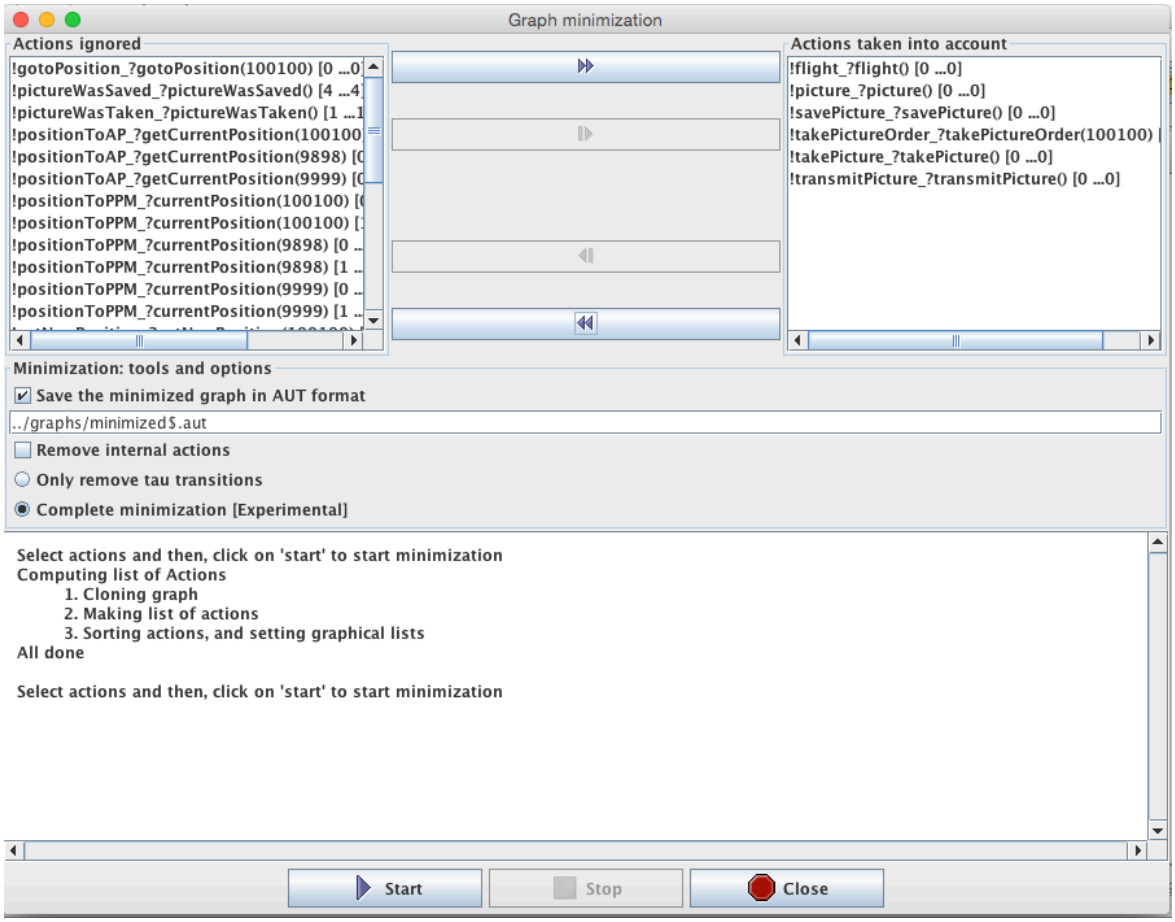

Figure 26. Observable Events Selection

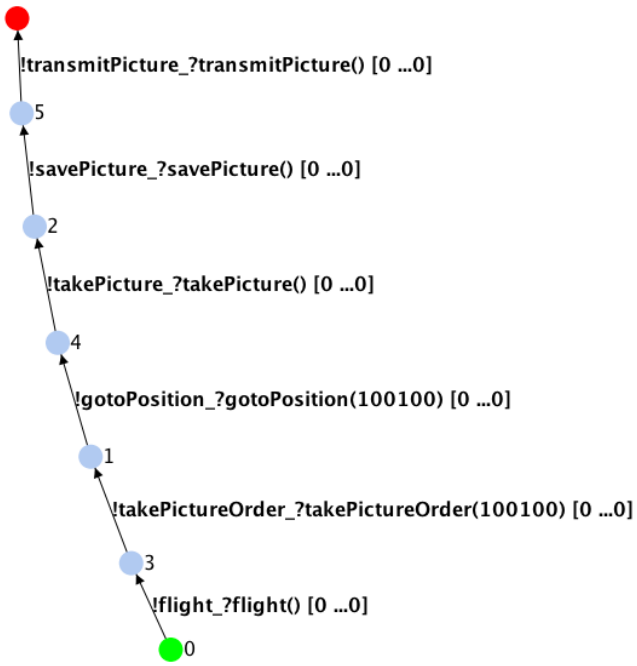

Figure 27. Quotient Automaton for Version 1 of the Model

Version 1 of the model may be safely saved. It is now ready to be extended to take degraded modes into account.

10. Degraded mode

Again, the method associated with SysML and TTool is incremental. In this paper, the UAV model has two versions, as shown by the <<versioning>> arrows

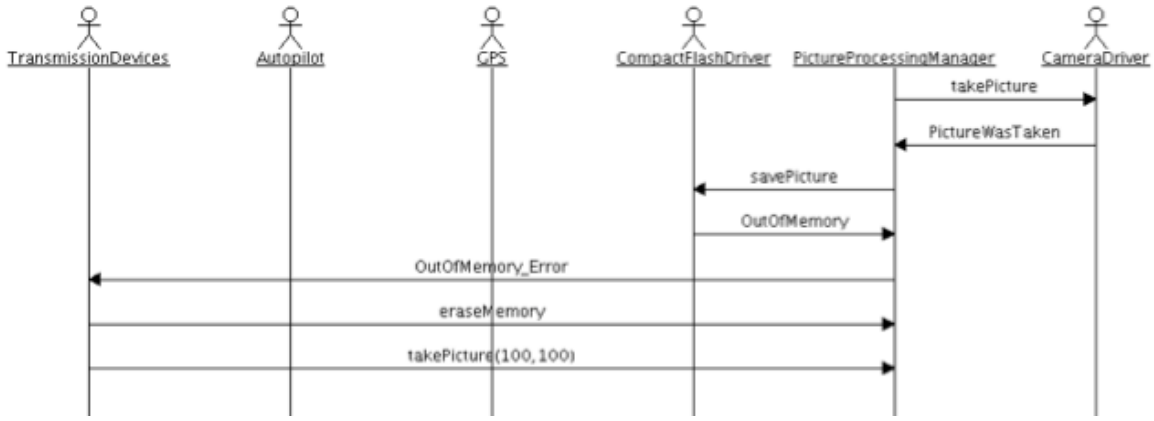

Figure 28. Degraded Scenario

The SysML model discussed in previous section is updated to include the scenario depicted by Figure 24. With the new block and state machine diagrams, the reachability graph has 446 states and 607 transitions. It is few to say the reachability graph is hard to exploit by hand (Figure 29).

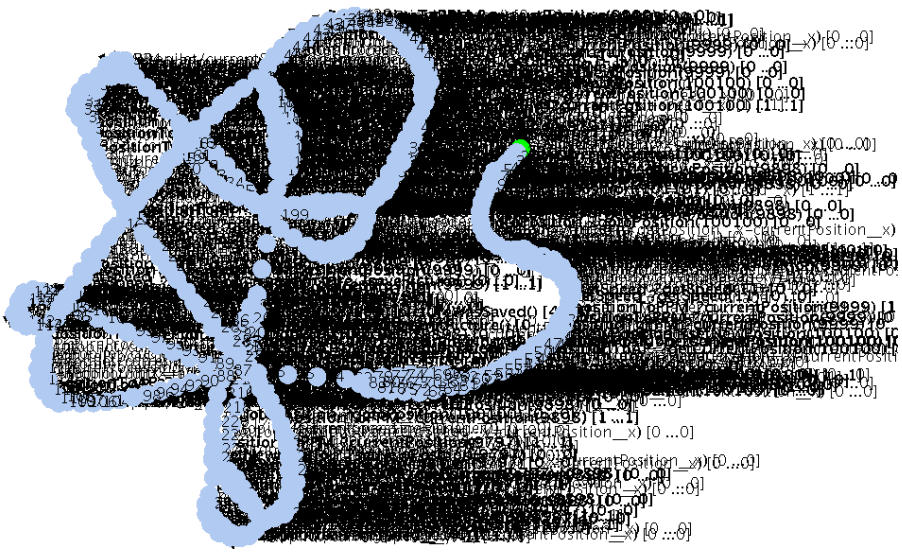

Figure 29. Reachability Graph

The reachability graph of Figure 29 is labeled by a set of events limited to orders from the ground station, taking pictures, and transmitting it. The resulting labeled transition system is minimized with respect to Milner's equivalence relation. Figure 30 depicts the quotient automaton (7 states, 10 transitions).

Unlike the reachability graph, the quotient automaton is easy to share and teamwork with. Figure 30 can be compared to Figure 27: version 2 of the SysML model conveys a more realistic usage of memory.

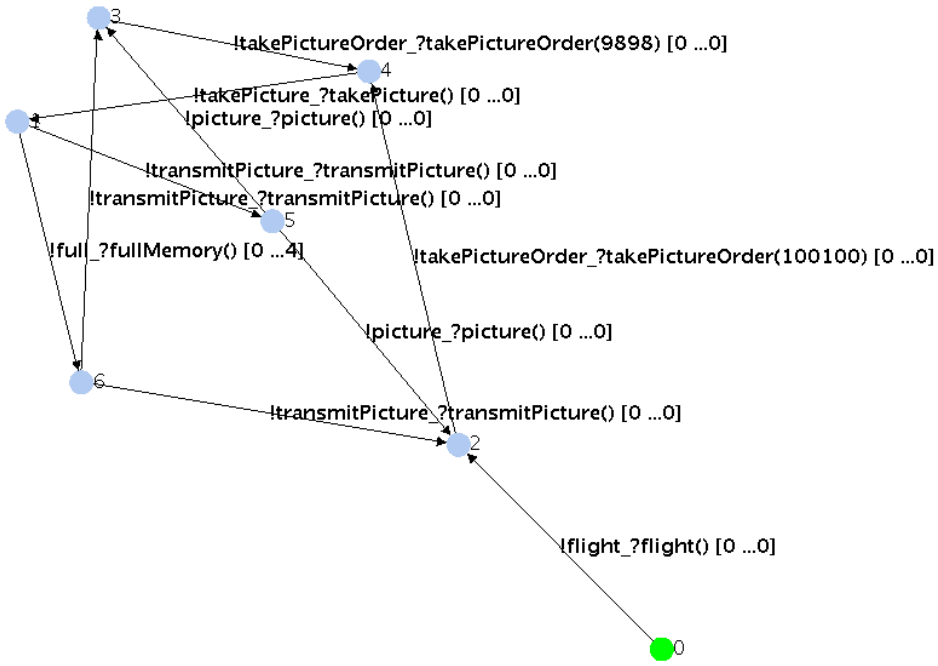

Figure 30. Quotient Automaton

11. Related Work

This section does not exhaustively lists SysML tools and focuses on the tools offering verification capabilities. It also mentions papers discussing application of SysML to UAVs.

SysML has two types of diagrams that can become executable ones: activity diagrams and state-machine diagrams. Activity diagrams are close to Petri nets and tools developed for the latter (e.g. [TINA, 2018]) can be reused in the context of SysML [Agarwal, 2013]. Intermediate languages different from Petri nets can be used [Ouchani, 2014]. On the other hand, state machine diagrams are contained in blocks and the block architecture can be seen as a communicating state machine composition via ports [Blondelle, 2015] [IFx-OMEGA, 2017] [Rhapsody, 2017].

In [Liu, 2010], Liu and Cao associate the SysML tool Rhapsody with Simulink to design a UAV flight control system. Rhapsody enables simulating the discrete/event part of the system behaviour but misses a continuous-time simulator: the authors to compensate that lack used Simulink. Accordingly the paper essentially discusses the use of blocks and state machine, and briefly surveys other diagrams used in the method associated with their tool.

In [Fernandez, 2016], Fernandez *et al*. use SysML not for designing a new system but for reengineering one.

7. Conclusions

Unmanned vehicles fall in the category of systems that capture complex design problems and question the benefits and potential of Model-Based System Engineering approaches. How to make a language, a tool and a method accepted by UAV designers is a really challenging issue.

The paper advocates for a MBSE approach based on SysML, the widely adopted standard throughout industry for system modeling. Unlike papers that limit the use of SysML to a industrial drawing, this paper proposes to use the free software TTool to consider a SysML model for early detection of design errors in the life cycle of a UAV. SysML models are debugged using the simulator of TTool and more systematically explored using formal verification techniques.

Simulation, verification and test generation are three important activities of the method proposed by the paper and associated with SysML and TTool. The method is entirely illustrated on a UAV in charge of taking pictures.

References

1. Agarwal, B., Transformation of UML Activity Diagrams into Petri Nets for Verification Purposes, International Journal of Engineering and Computer Science, vol. 2, no.3, pp. 798-805.
2. Andersson, H, Herzog, E, Johansson G., Johansson, O., Experience from introducing Unified Modeling Language/Systems Modeling Language at Saab Aerosystems, Vol. 13, Issue 4, Winter 2010, pp. 369-380.
3. Apvrille, L., Saqui-Sannes, P. de, Requirements Analysis, Book chapter in Embedded Systems: Analysis and Modeling with SysML, UML and AADL, Edited by F. Kordon, J. Hugues, A. Canals and A. Dohet, Ed. ISTE /Wiley, May 20, 2013, ISBN-13: 978 1848215009.
4. Arantes, M., Bonnard R., Mattei A.-P., Saqui-Sannes P. de, General Architecture for Data Analysis in Industry 4.0 using SysML and Model Based System Engineering, 12th Annual IEEE International Systems Conference (SysCon 2018), April 2018, Vancouver, BC, Canada.
5. Baduel, R., Chami, M., Bruel J.-M., Ober I., SysML Models Verification and Validation in an Industrial Context: Challenges and Experimentation, European Conference on Modeling Foundations and Applications (ECMFA 2018), LNCS, volume 10890, pp. 132-146.
6. Blondelle, G., Bordeleau, F., Exertier, D., Polarsys: A New Collaborative Ecosystem for Open Source Solutions for Systems Engineering Driven by Major Industry Players, Insigight, Vol. 18, no.2, August 2015, pp. 35-38.
7. Courtiat, J.-P., Ayache, J.-M., Algayres, B., Petri Nets are Good for Protocols, Computer Communication Review, 1984.
8. Courtiat, J.-P., Saqui-Sannes, P. de, ESTIM: an Integrated Environment for the Simulation and Verification of OSI Protocols Specified in Estelle, Computer Networks and ISDN Systems, Vol. 25, no.1, August 1992, pp. 83-98.
9. Deitsh, A, Schneider, V., Kane, J., Dulz, W., German, R., Towards an Efficient High-Level Modeling of Heterogeneous Image Processing Systems, 2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS), Pasadena, CA, 2016, pp. 1-6.
10. Dssouli, R, Khoumsi, A., Elqortobi M., Bentabar. J., Chapter Three - Testing the Control-Flow, Data-Flow, and Time Aspects of Communication Systems: A Survey. Advances in Computers 106: 95-155 (2017).
11. Fernandez J, Lopez, J., Patricio Gomez, J, Reengineering the Avionics of an Unmanned Aerial Vehicle, IEEE Aerospace and Electronic Systems Magazine, vol. 31, no. 4, pp. 6-13, April 2016.
12. Friedenthal S, Moore A, Steiner R., A Practical Guide to SysML, Second Edition,OMG Waltham, Massachusetts: Elsevier, 2012.
13. Fitzgerald, J., Gamble, C., Gorm Larsen, P., Pierce, K., Woodcock, J. (2015), Cyber-physical systems design: foundations, methods and integrated tool chains, IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE), Florence, Italy, pp. 40-46.
14. IEEE, 1012-2012 Standard for Systems and Software Verification and Validation, June 2005.
15. Ifx-OMEGA, https://www.irit.fr/ifx/, last accessed July 2017.
16. INCOSE Handbook V4, https://www.incose.org/products-and-publications/se-handbook, July 2015.
17. INCOSE, Systems Engineering Vision 2020, version 2.03 edn, Technical Operations, International Council on Systems Engineering, September 2017.
18. X. h. Liu and Y. f. Cao, Design of UAV Flight Control System Virtual prototype using Rhapsody and Simulink, 2010 International Conference On Computer Design and Applications, Qinhuangdao, 2010, pp. V3-34-V3-38.
19. Kayal, I., Farid, A., The Need for Systems Tools in the Practical Of Clinical Medicine, Systems Engineering,
20. Le Sergent, T, SCADE: A Comprehensive Framework for Critical System and Software Engineering, International SDL Forum 2011, LNCS, volume 7083.
21. Madni, A., Sievers, M., Model-based systems engineering: Motivation, current status, and research opportunities, Systems Engineering, May 2018.

22. Mattei, A-P., Loures, L., Saqui-Sannes, P. de, Escudier, B., Feasibility study of a multispectral camera with automatic processing onboard a 27U satellite using Model Based Space System Engineering, IEEE Systems Conference April 2017, Montreal, Qc, Canada.
23. Milner, R., A Calculus of Communicating Systems, LNCS 92, Springer-Verlag, 1980.
24. NASA Systems Engineering Handbook, NASA/SP-2007-6105 Rev 1, 2007.
25. OMG, UML, Unified Modeling Language, December 2017, https://www.omg.org/spec/UML/2.5.1/.
26. OMG, SysML, Systems Modeling Language 1.5, May 2017, http://www.omg.org/spec/SysML/1.5/.
27. Ouchani, S., Aït Mohamed, O., Debbabi, M., 2014, A formal verification framework for SysML activity diagrams, Expert Systems with Applications, (41) 6, pp. 2713-2728.
28. Rhapsody, https://www.ibm.com/us-en/marketplace/architect-for-systems-engineers, last accessed July 2017.
29. Saqui-Sannes, P. de, Apvrille, L., 2016, Making Modeling Assumptions an Explicit Part of Real-Time Systems Models", 8th European Congress on Embedded Real Time Software and Systems (ERTS), Toulouse, France, pp. 27-29
30. Saqui-Sannes, P. de, Vingerhoeds, R., Apvrille, L, EarlyChecking of SysML Models applied to protocols, 12th International Conference on Modeling, Optimisation and Simulation (Mosim 2018), June 2018, Toulouse, France.
31. SEBOK, Guide to the Systems Engineering Body of Knowledge, V1.9, 2018.
32. TINA, http://projects.laas.fr/tina/, last accessed September 2018.
33. TTool, https://ttool.telecom-paristech.fr/, last accessed September 2018.
34. UPPAAL, http://www.uppaal.org/, last accessed September 2018.
35. Vernadat, F., Percebois, c., Farail, P., Vingerhoeds, R., Rossignon, Alain, Talpin J.-P., Chemouil D., The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development, Data Systems In Aerospace (DASIA 2006), Berlin, Germany, ESA, May 2006.
36. Wassem, M., Usaman Adiq, M., Application of Model-Based Systems Engineering in Small Satellite Conceptual Design: A SysML Approach, IEEE Aerospace and Electronic Systems Magazine, Vol. 33, Issue 4, April 2018.