A new UML tool-based Methodology for the Software Requirements Analysis

Thomas Weyrath, Franz Schöttl, Berthold Schinnerl, Herbert Schreyer

A new UML tool-based Methodology for the Software Requirements Analysis

Thomas Weyrath
Elektroniksystem- und Logistik-GmbH
Email: Thomas.Weyrath@eurocopter.com

Berthold Schinnerl
German Federal Armed Forces
Email: Berthold.Schinnerl@eurocopter.com

Franz Schöttl
Elektroniksystem- und Logistik-GmbH
Email: Franz.Schoettl@eurocopter.com

Herbert Schreyer
Eurocopter Deutschland GmbH
Email: Herbert.Schreyer@eurocopter.com

Abstract

Current standards in avionic (e.g. DO-178C) and automotive (e.g. ISO 26262) address model-based software engineering techniques. Hence, it is obvious to deal with these techniques methodically in the corresponding development processes. This article describes a new UML tool-based methodology to create the software requirements analysis. The methodology is use case driven and uses natural language requirements as well as UML diagrams. The later stages of the development process, e.g. software design, are not covered in this paper.

Keywords: UML, Methodology, Software Requirements Specification.

1 Introduction

The System Support Centre NH90/Tiger (SSC) is a cooperative organisation of German Federal Armed Forces, Eurocopter, ESG, Cassidian and MBDA. It comprises a team of 200 engineers. Its main task is the avionic software maintenance and modification of the military helicopters NH90 and UH Tiger. The maintenance comprises more than 12.000.000 lines of code in 16 avionic computers, including the ground support system. Additionally, the documentation of the development is immense. It contains over 500 documents of specifications and design descriptions. Most of them have hundreds of pages, some of them over a thousand. In view of the fact that the helicopters have a life cycle of more than 40 years, the software maintenance and modification is a challenge.

Another challenge is the obsolescence problem occurring with used tools, databases and formats of electronic documents. The avionic software of both helicopters was developed in the 1990s and it is difficult to keep the expertise of the past and present.

1.1 What are the objectives of SSC?

Over the years the helicopter software will be changed due to modifications in the software environment, new customer requirements or simply by fixing errors. These changes will have to be done efficiently and reliably. This includes that all state of the art processes and methodologies of software maintenance and modification have to be well known. These processes have to be improved continuously and adequate software tools have to be applied. Hence, it has to be ensured that the software stays long-term maintainable because of the long life cycle of the helicopters. If the maintainability is not given, it has to be established.

1.2 Why does the SSC use UML?

UML is standardized [15] and defines a set of graphical notation techniques that are well documented. It supports the application of well-accepted software concepts, like modularization, structuring and abstraction. These concepts improve the understanding of the system and help (new) employees to familiarize themselves with the helicopter software (see [14]).

In addition the SSC wants to introduce UML for the following reasons:

1. As regards the Tiger: UML could help to avoid the obsolescence problem because most of the Tiger specifications based on Structured Analysis (SA), a method that was state of the art in the 1980s. The tools are no longer supported and no appropriate successors exist.

2. As regards the NH90: Most of the existing NH90 specifications are text-based and use non-standardized graphics to illustrate the textual requirements only. The use of UML as a formal method should a) harmonize the different graphic styles and b) increase the understanding of the software functionalities.

With the help of external consultants, the SSC has established a customized UML methodology for software requirements analysis and design, which refines an EADS

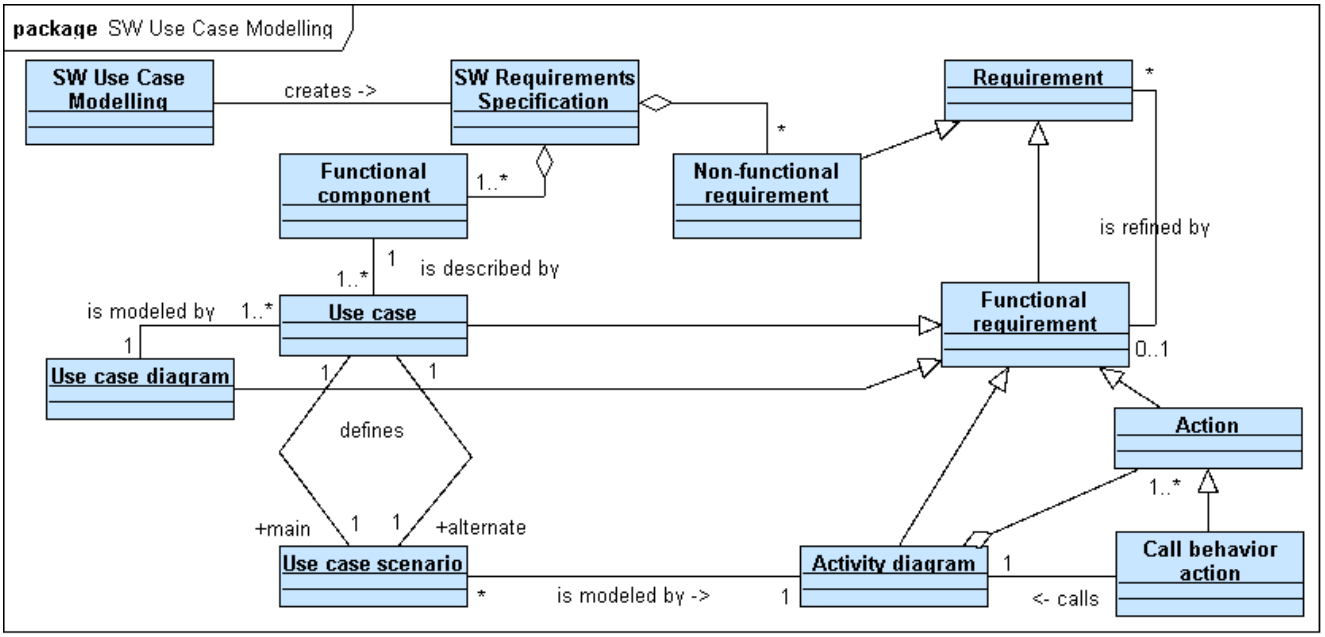

Figure 1. Meta model of the software use case modelling.

guideline for using UML (see [11]). After this, those consultants have trained SSC employees in both UML and UML methodology. In addition, the methodology has been successfully applied in the context of a redesign of an embedded avionic computer. Furthermore, it has been in several software projects of the SSC that aim at improving the testing environment of the airborne computers.

2 Software Requirements Analysis

The objective of the software requirements analysis is to elicit stakeholders needs and expectations (in particular: users and customers), *that constitute an understanding of what will satisfy the stakeholders* [1]. This outcome leads to high-level requirements that describe the software system from customer (or black box) view.

Since the software requirements analysis is a difficult process, the methodology of the SSC breaks it down into two smaller, well-defined steps: the software use case modelling and the software domain analysis modelling.

The methodology is exemplarily shown by modelling a simplified flight management system as described in [19].

2.1 Software Use Case Modelling

The software use case modelling establishes the system's functional requirements. It provides a way for engineers and developers to get to a common understanding with customers and users.

Figure 1 shows a summary of the key concepts used in the software use case modelling. Key concepts are the use cases, use case diagrams, actions and activity diagrams. The activity diagrams model the use case scenarios. These key concepts modell functional requirements, which can be translated in natural language requirements.

2.1.1 Step 1: Divide the software system into functional components

Avionic software systems are very complex. Hence, there is a need to divide their functionality into functional components (or functional partition, see [1]) to handle the complexity of the model. This is analogous to the decomposition of the software system into software components and -modules later in the software design modelling.

The customized methodology of SSC for the modelling of this functional decomposition is:

1. Create a package with name "Functions of the System" and provide a brief description of the system in the documentation field of this package.
2. For each identified functional component ⟨FC⟩ do:
   (a) Create a package with name ⟨FC⟩ and describe the task of ⟨FC⟩ in its documentation field.
   (b) Perform a use case analysis (see step 2).
3. Create a class diagram with name "Functions of the System" to visualize the functional components as packages.

Figure 2 shows the class diagram to model the functional components of the example flight management system. The system is divided into the functional components *navigation*, *flight planning*, *trajectory predictions*, *performance computations*, and *guidance* (see [19]). The grey filled package shows the parts, which are refined in the example.

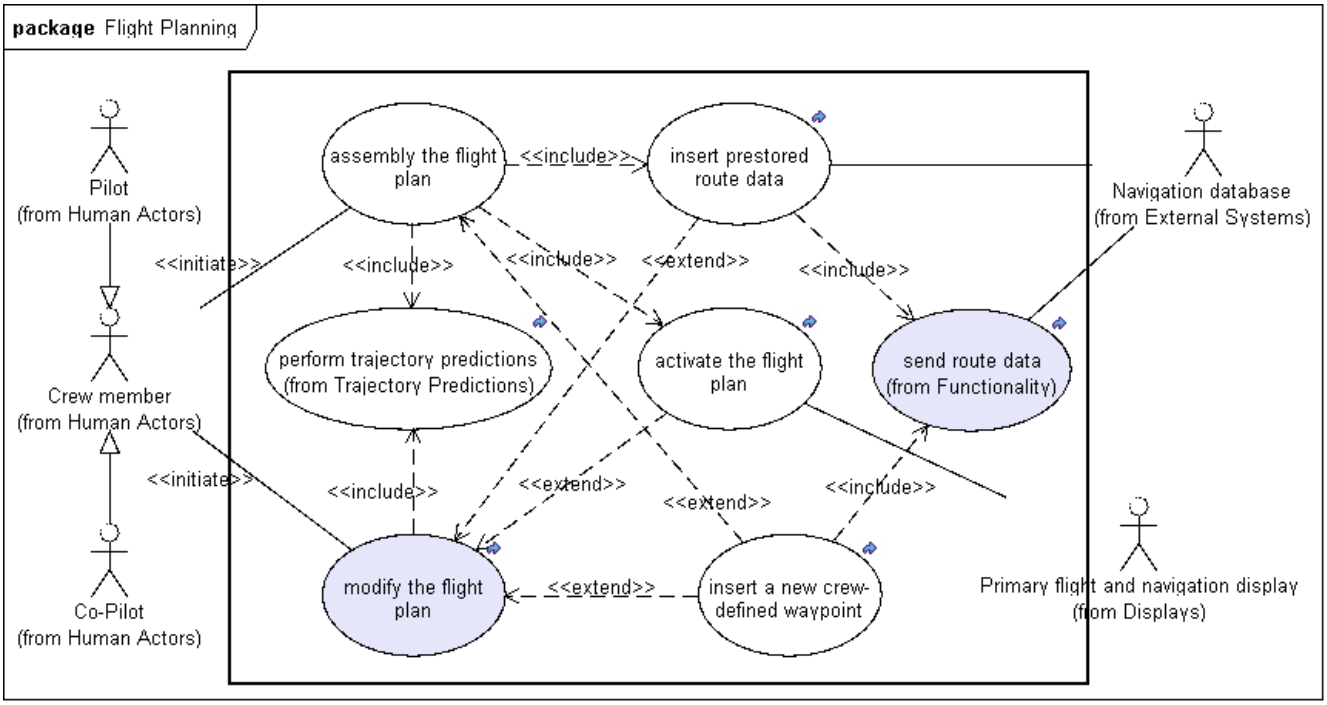

Figure 3. Use case diagram for the flight planning function.

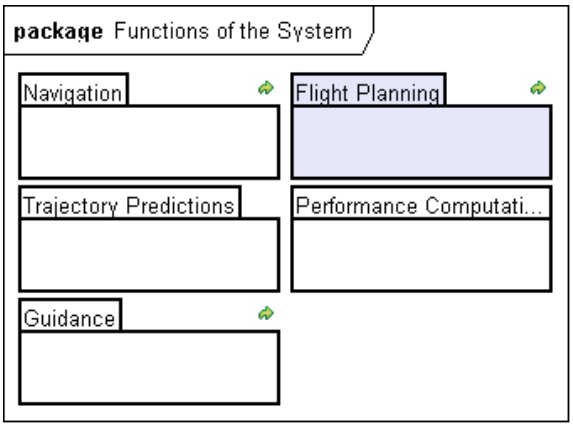

Figure 2. Package "Functions of the System".

2.1.2 Step 2: Perform a use case analysis

After the functional decomposition of the software system (step 1), a use case analysis is used to identify its requirements.

Use cases *capture system functionality and requirements* [16] of a software system. They can be seen as *a contract between the stakeholders of a system about its behavior* [5, 6]. Each use case specifies *a set of actions to yield an observable result of value to an actor* [4, 18]. It describes the functionality from a black box view. That means that the system is determined by describing its behavior. Only its inputs and outputs, respectively the interactions with external actors, are considered without seeing the inner working of the system.

The customized methodology of SSC for the use case analysis is:

1. Identify use cases and actors by name and provide a brief description in the corresponding documentation fields (see step 2a).
2. Create the use cases and group them under[1] the packages of the functional components (from step 1).
3. For each functional component ⟨FC⟩ create a use case diagram with name ⟨FC⟩ in the package of ⟨FC⟩ and structure its use cases graphically (see step 2b).
4. For each use case ⟨UC⟩ do:
   (a) Create a textual description of ⟨UC⟩ (see step 2c).
   (b) Create one activity diagram ⟨AC⟩ with name ⟨UC⟩ under ⟨UC⟩. The activity diagram ⟨AC⟩ models the main scenario (incl. alternative scenarios) of ⟨UC⟩ (see step 3).
5. If ⟨AC⟩ does not model the use case complete then create further requirements (functional or non-functional) that enrich ⟨AC⟩ and its actions with further specifics (see step 6).

Step 2a: Identify use cases and actors
Finding use cases is often not easy. One way is to look for steps in the business processes that the system has to support. Another way is to identify the main functionalities of the system, i.e. *the reason for using the system*

[1] Because of the model has a tree structure, "under" means "as children of".

[13]. Figure 3 shows the use cases of the flight planning function (from [19]). The diagram is illustrated exemplarily by the use case "modify the flight plan". [19] serves as a higher-level system specification: *"The flight plan modification can come from crew selections [...]"*, i.e. "modify the flight plan" is initiated by the actor "crew-member". "The flight plan is *constructed by linking data stored in the navigation database [...]"*, i.e. it is extended by the use case "insert pre-stored route data". *"[...] When the desired changes have been made [...] this modified flight plan is activated by the crew."*, i.e. the use case is extended by "activate the flight plan".

A use case must be initiated by someone or something outside the scope of the use case [16]. This is called an actor. An actor does not need to be a human. It can be any external system. All actors should be defined centrally in the package "Actors" of the Domain model (for details, see step 4).

Step 2b: Structure the use cases graphically
The use case diagram structures the use cases graphically and describes the relations between use cases, actors and the system.

The customized methodology of SSC is:

1. Draw a system boundary box. The boundary box indicates the scope of the system (the subject) and *anything not realized by the subject is considered outside the system boundaries and should be modeled as an actor* [16].

2. For each use case ⟨UC⟩ identified in step 2a do:

   (a) Draw the use case with name ⟨UC⟩ inside the boundary box .

   (b) If an actor with name ⟨U⟩ is involved in ⟨UC⟩ (active or passive) then draw the actor ⟨U⟩ (from package "Actors") outside of the boundary box and draw an association to ⟨UC⟩.

   (c) If ⟨U⟩ triggers or initiate ⟨UC⟩, add the stereotype ⟨⟨initiate⟩⟩ to the association.

3. If the behavior of use case ⟨UC1⟩ is adopted completely in another use case ⟨UC2⟩ then use the ⟨⟨include⟩⟩ relationship (⟨UC2⟩→⟨UC1⟩).

4. If use case ⟨UC1⟩ extends another use case ⟨UC2⟩ then use the ⟨⟨extend⟩⟩ relationship (⟨UC1⟩→⟨UC2⟩). The extend relationship shall include the condition that must be satisfied and reference to the extension points which defines the locations in the extended ⟨UC2⟩.

Figure 3 shows the result of the use case diagram for the functional component "Flight Planning" of the example system.

Step 2c: Describe use cases textually
The use case description shall give a raw (but valid) survey about the functional goal. It is described by the following list of properties.[2]

Use case name	modify the flight plan
Purpose	The flight plan has to be modified because of crew selections. The use case does not consider that the flight plan is modified via data link. The modified flight plan is stored in the navigation database.
Actors	crew member, navigation database
Trigger	The crew selects the command to modify the flight plan.
Preconditions	1. A flight plan exists and is activated.
Basic scenario	1. The system creates a modified (temporary) copy of the active flight plan. 2. The crew does changes in the flight plan (insert / edit / delete route data). 3. The system performs the trajectory predictions on the modified flight plan. 4. The crew activate the modified flight plan. 5. The use case ends.
Alternative scenarios	4.a. If the changes should be dismissed then 4.a.1. The crew aborts the modification. 4.a.2. The system closes the flight plan modification page. 4.a.2. Go to 5
Postconditions	1. The modified flight plan is activated. 2. The modified flight plan is stored in the navigation database. 3. The modified flight plan is displayed on the primary flight display.
Failures	If the flight database is full then an error message occurs.
Non-functional	The crew can modify the flight plan at any time.

Table 1. Example of a use case description

Use case name see step 2a.

Actors list of actors that are involved, see step 2a.

Purpose describes, what the user intends to achieve with the use case.

Trigger describes the event that initiates the use case.

Preconditions must be true before initiating the use case.

Basic scenario defines the typical scenario (the most common case) as a sequence of enumerated actions, e.g.

1. `<action 1>`

2. `<action 2>`

3. `<action 3> [...]`

4. `the use case ends.`

[2] In the methodology of SSC these properties are captured as "free text" entries in the TOPCASED requirement module that are attached to the use case.

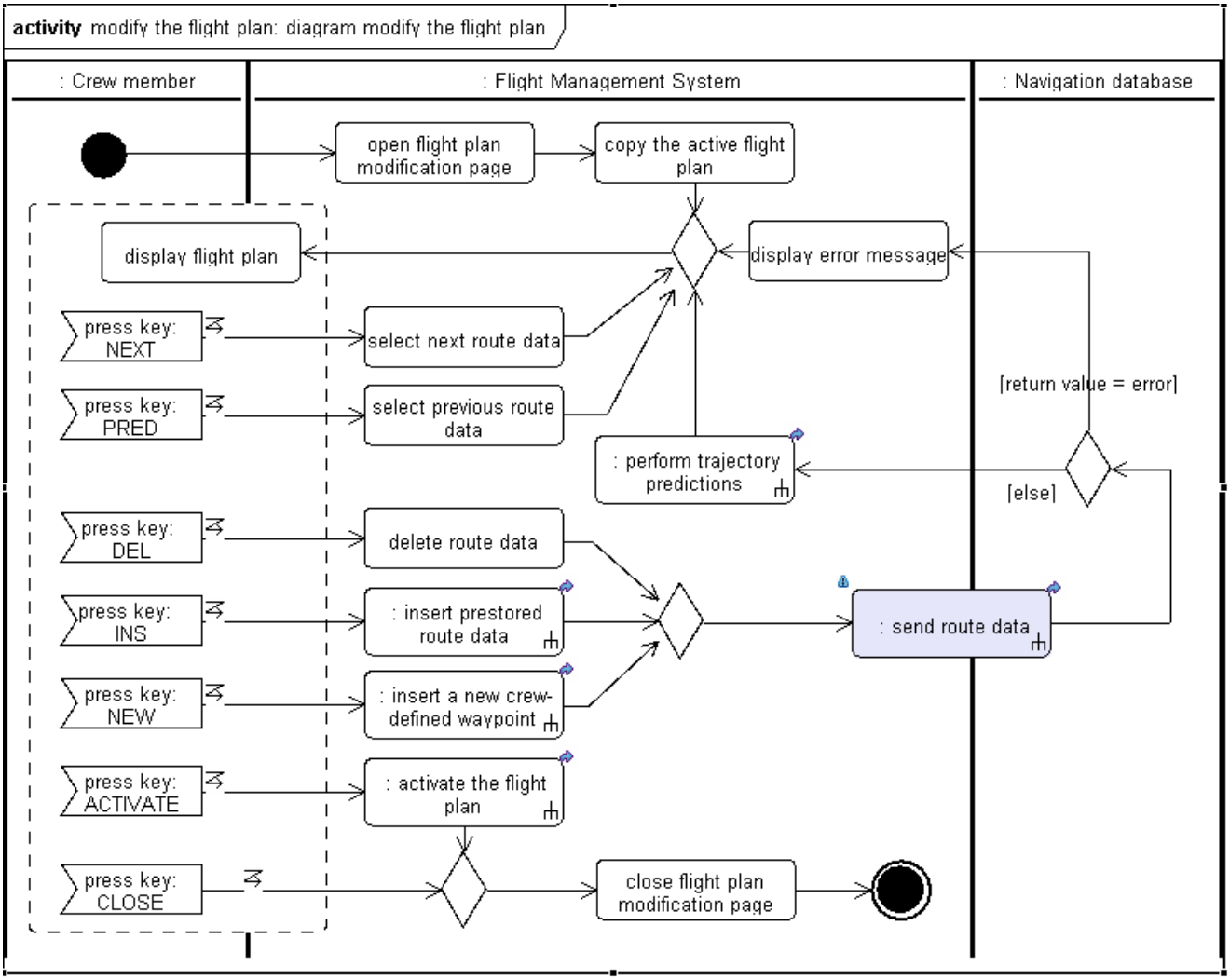

Figure 4. Activity diagram for the use case "modify the flight plan".

Note: If the use case terminates, the last action is called "the use case ends". Otherwise, it means that the use case does not terminate and runs endless.

Alternative scenarios are variants that differ from the basic scenario in form of conditional statements as sequence of enumerated, prefixed actions, e.g.

2.a. `if <condition> then`

2.a.1. `<action a1>`

2.a.2. `<action a2> [...]`

2.a.3. `go to 4`

Postconditions occur i) when the use case is successfully completed and ii) when the use case has been terminated due to an exceptional condition.

Failures describe exceptional conditions of the use case and its processing.

Non-functional specify constraints of the use case (e.g. timing conditions).

The table 1 shows an example for the description of the use case "modify the flight plan".

2.1.3 Step 3: Model an activity diagram

Use case descriptions do not replace formal requirements, because they are too raw and maybe incomplete. To develop requirements, each use case should be modelled by an activity diagram that describes the main scenario functionally and completely including the variants and the failure processing.

The customized methodology of SSC is:

1. For each involved actor $\langle U \rangle$ and the system $\langle S \rangle$ create a swim lane. The swim lanes of $\langle U \rangle$ is represented by the corresponding actor of the domain model, respectively the swim lane of $\langle S \rangle$ by the corresponding class.

2. Start the activity diagram with an initial node. After that, continue with the first action of the scenario. The trigger of the use case is not modelled. Instead of that a requirement for the trigger is attached to the activity or

(if the trigger is more complex) an additional activity diagram is modelled in the package ”Trigger”.

3. For each action ⟨A⟩ of the basic scenario and the alternative scenarios, put an action with name ⟨A⟩ in the corresponding swim lane. More precisely:

   (a) Place a *user interaction* into the swim lane of the user, e.g. ”press key: NEXT” in figure 4. This corresponds to the natural language requirement: *”The system shall provide the user with the ability to press key NEXT.”*

   (b) Place an *internal system action* into the swim lane of the system, e.g. ”select next route data” in figure 4. This corresponds to the natural language requirement: *”The system shall select the next route data.”*

   (c) Place a *system action to the user* on the line between the human actor and system, e.g. ”display flight plan” in figure 4. This corresponds to the natural language requirement: *”The system shall display the flight plan to the crew member.”*

   (d) Place an *interface action to an external system* on the line between the system and the external system, e.g. ”send route data” in figure 4. This corresponds to the natural language requirement: *”The system shall be able to send the route data to the navigation database.”*

4. Special case: for modelling user interactions put an interruptible activity region in the user’s swim lane. All user interactions are modelled as event actions and are connected with system actions outside the region. The system action that can be interrupted (e.g. ”display”, ”wait”) should be put in this region.

5. Special case: each included or extended use case is represented by a call behavior action whose behavior is linked with the activity of the corresponding use case.

6. For each variant that branches from the basic course put a decision node and label the branches from the decision node to the following actions with conditional statements. If variants come back to the basic course, connect them with a merge node.

7. Model concurrent actions using join nodes and fork nodes.

8. Finalize the activity diagram with one or more final node.

As shown, each action – together with his connections to other actions – can be translated into a corresponding natural language requirement. Hence, the activity diagram itself represents all these requirements altogether. Figure 4 shows the activity diagram for the use case ”modify the flight plan”.

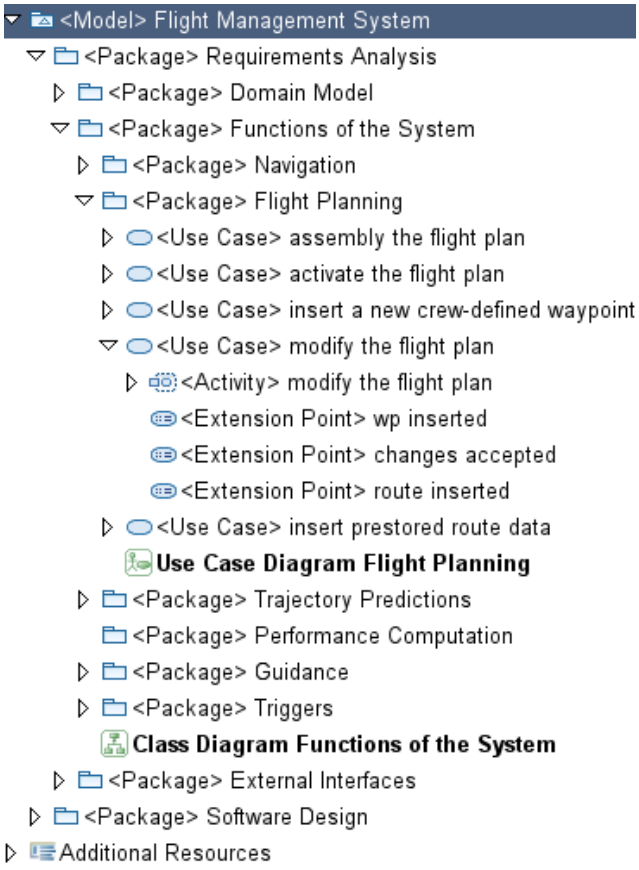

Figure 5. Model structure.

2.1.4 Structuring the Model

The customized methodology of the SSC defines the main structure of the UML model. The reason for this is that the SRS document generator (see section 3) needs in part a fixed model structure. In addtion, the comparable model structure improves the readability and the understanding.

Figure 5 shows a part of the model structure of the requirements analysis. It consists basically of the packages ”Domain Model”, ”Functions of the System”, and ”External Interfaces”.

2.2 Software Domain Analysis Modelling

The software domain analysis modelling refines the results of the use case analysis to high-level requirements that are correct, unambiguous, complete, consistent, verifiable, modifiable and traceable [3, 17].

The software domain analysis is performed as an expanded use case analysis with focus on a functional approach (not as an object-oriented analysis). For the SSC the functional aspects of the software are most important and this is closer to the used development standards, DOD-2167A and DO178(A/B) ([7]).

Figure 6 shows a summery of the key concepts used in the software domain analysis modelling. The key concepts are the domain model (consisting of actors, terms and data

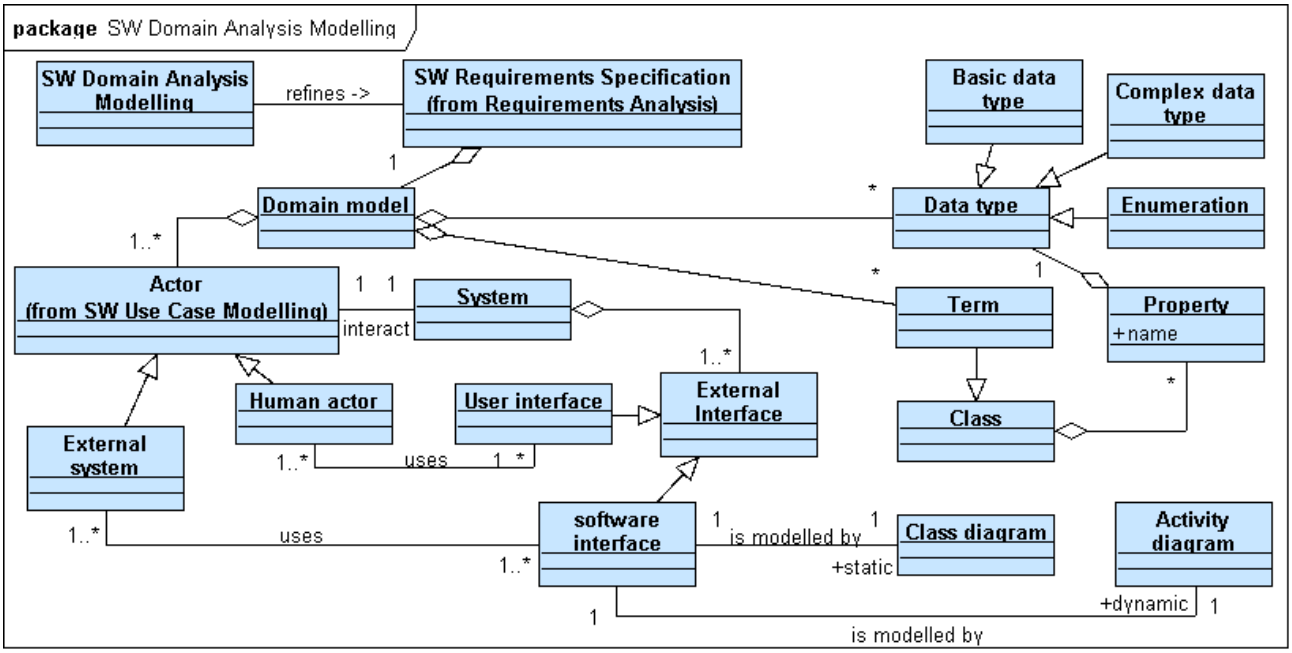

Figure 6. Meta model of the software domain analysis modelling.

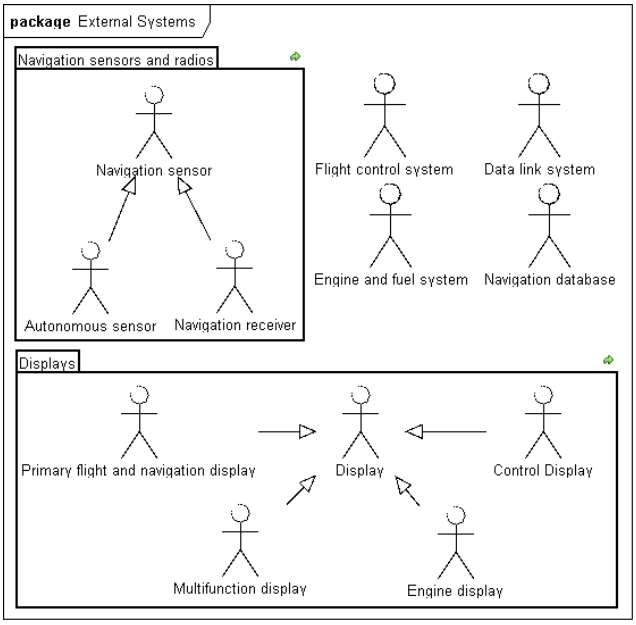

Figure 7. Domain model: external (avionic) systems.

types) and the external interfaces, especially the software interfaces (consisting of the static and dynamic descriptions).

2.2.1 Step 4: Create a domain model

The domain model describes the vocabulary and the domain specific knowledge of the problem domain. It consists of the following parts:

Actors consist of all human actors (e.g. users) of the system and external systems that communicate with the system.

Terms define the technical terminology and business concepts, e.g. tangible objects, roles that objects play, objects that contain other objects, processes, events.

Primitive types define predefined data types, e.g. boolean, int, float.

Enumerations and composite types define sets of named values and composite types that are derived from more than one primitive type .

Process words define the meaning of often-used verbs, e.g. store, display, send.

The customized methodology of SSC to create the domain model is:

1. Create a package ”Domain Model” that contains all the following packages.
2. Create a package ”Actors” and a use case diagram with all identified actors (see example in figure 7).
3. Create a package ”Primitive types” and a class diagram with all required primitive types.
4. Create a package ”Enumerations and composite types” and a class diagram.
   (a) For each set of named values create an enumeration and add each value as enumeration literal.
   (b) For each composite type create a data type and add each value as property with a primitive type or another composite type.

5. Create a package ”Terms” and a class diagram. For each identified term create a class and add attributes as property with an enumeration, primitive or composite type (see example in figure 8).

6. Model relationships between the terms as association, generalization, aggregation or composition. Name each association with either an association name or role names or both and add it with multiplicities.

7. Create a package ”Process words” and a class diagram. For each identified process word create a class.

8. Give a textual definition of all terms, data types and actors in the corresponding documentation field.

It is important that all used entities are defined and that *each characteristic [of the software systems] should be described using a single unique term* [3].

All textual definitions should be written using sentence patterns (see [17]). The pattern for terms and actors is: *”In the system ⟨name⟩ ⟨term⟩ shall be defined as ⟨explanation⟩.”* The pattern for process word is: ”In the system ⟨name⟩ to ⟨process word⟩ shall be defined as the process of ⟨explanation⟩.” Synonyms (e.g. ”to save” and ”to store”) should be defined, too.

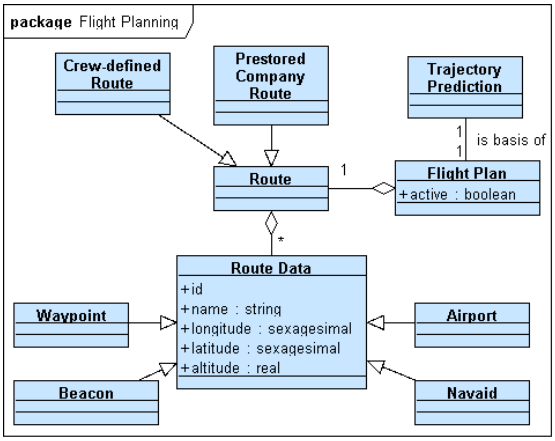

Figure 8. Domain model: Terms.

2.2.2 Step 5: Model external interfaces

The external interface description in the SRS addresses *how the software interact with people, the system's hardware, other hardware, and other software*[3]. The interface modelling is limited to the software interfaces that describe the data inputs and outputs between the software system itself and its external software systems.

The customized methodology of SSC is:

1. For each external software system ⟨X⟩ with a software interface create a package with name ⟨X⟩. The external software system shall be already defined as actor in the domain model.

2. For each identified interface ⟨I⟩ create a package with name ⟨I⟩. Provide a brief description of the interface in the package documentation field.

3. In the package ⟨I⟩ create the package ”Input/Output” and describe the static aspects of the interface:

   (a) Create a class diagram with name ⟨I⟩

   (b) Create two new classes with name ⟨I⟩+”Input data” and ⟨I⟩+”Output data”.

   (c) Put all classes from the domain model (terms) in this diagram, that describe the data exchange between the system and the external system and assign them to the classes ”input / output data” via aggregation depending on whether they are inputs or outputs of the system.

4. In the package ⟨X⟩ create the package ”Functionality” and describe the dynamic aspects of the interface:

   (a) Create a use case diagram and a use case with name ⟨I⟩ that define the interface functionality.

   (b) Draw the actor of the external system from package ”Actors” and assign it with an association.

   (c) Create a simple activity diagram ⟨AC⟩ with name ⟨I⟩ under the use case.

   (d) Put two activity parameter in the activity diagram ⟨AC⟩ and assign them to the input (respectively output) classes.

   (e) If needed add timing requirements, e.g. in form of accept time event action.

5. Put the interface activity as call behavior action' in the activity diagram of the use case.

Figure 9 shows the model of the external interface description ”send rout data” to the navigation database. The activity consists of an signal action ”send data”, which is connected to the input activity parameter via an input pin. After the route data have been sent, The event action ”receive return value” waits for the answer of the navigation database. The result value is returned to the output activity parameter via the output pin. Concurrently, after timeout an error value is returned.

2.2.3 Step 6: Develop natural language requirements

Not all requirements can be modelled with UML. Either the requirement is too involved to model or the requirement has a non-functional character (see [12]). For this reason, it is necessary to develop natural language requirements in parallel to the UML model. TOPCASED provides a way to create textual requirements that refine the model elements (see [9]).

Similar to the textual definitions all textual requirements should be written using the following sentence pattern (see [17]):

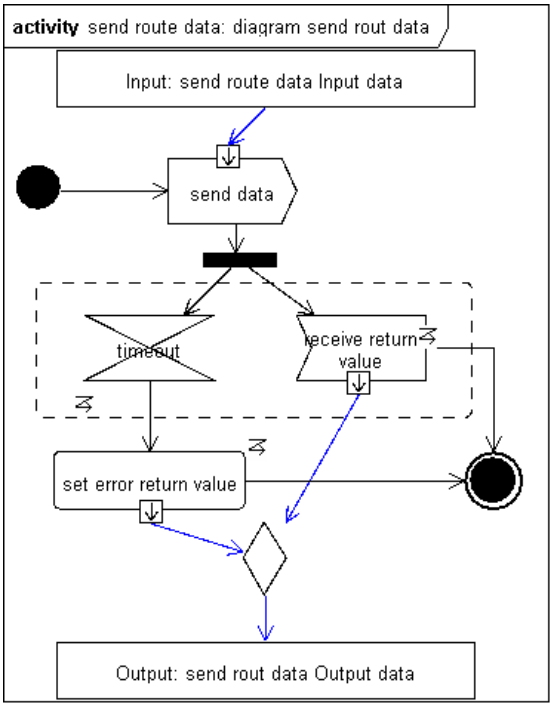

Figure 9. Activity diagram "send route data".

- For a system activtiy: "[⟨When?⟩ ⟨Under what conditions?⟩] the system ⟨name⟩ shall ⟨process⟩ ⟨object⟩ [⟨additional details about the object⟩]."
- For an user interaction: "[⟨When?⟩ ⟨Under what conditions?⟩] the system ⟨name⟩ shall provide ⟨whom?⟩ with the ability to ⟨process⟩ ⟨object⟩ [⟨additional details about the object⟩]."
- For an interface requirements: "[⟨When?⟩ ⟨Under what conditions?⟩] the system ⟨name⟩ shall be able to ⟨process⟩ ⟨object⟩ [⟨additional details about the object⟩] from ⟨external system⟩."

Section 2.1.3 contains some examples of textual requirements.

3 Generating the Software Requirements Specification (SRS)

The generation of the SRS forms the conclusion of the software requirements analysis. Every UML tool should provide a method to generate the documentation of the model, which has been built. Within the SSC, the UML tool TOPCASED is used. TOPCASED includes the document generation Gendoc2 [10], which uses the model to text transformation language Acceleo.

The creation of the document generator includes the following actions (see [2]):

1. Create a template for the SRS document according to the used development standards, which determines the content and structure of the development documents.
2. Insert a script part for the model specific content, i.e.:
   (a) The template configuration ⟨config⟩ for the input model and the output documentation path.
   (b) The script context ⟨context⟩ for the model absolute path, the path to the model element to start (optional) and the list of external bundles to import.
   (c) The script content ⟨gendoc⟩ for headings, descriptions (documentation fields), diagrams as images and textual requirements.
3. Adapt the output parameter with the own project parameters (model name, output file) in the template header and generate the document.

The following script snippet shows an extract of the SRS template.

```
3.1 Actors
<context
  model = '${model}'
  element = '[...]/Domain Model/Actors'
  importedBundles = [...]/>
<gendoc>
  [for (p:Package|self.ownedElement)]
3.1.1 [p.name/]
    [if (p.getAllDiagrams()->size()>0)]
      [p.name/]
      [for (s : String|
        splitNewLine(p.getDocumentation()))]
        [s/]
      [/for]
      <image object=[self.getDiagram()/]
        keepW=true>
        [p.name/]
      </image>
    [/if]
  [/for]
</gendoc>
```

4 Training

UML is a language. This means it has both, syntax and semantics. [. . .] there are rules regarding how the elements can be put together and what it means when they are organized in a certain way. [16]. But UML does not provide any procedures or methods how models should be created and which diagrams should be used to build a clear and understandable model of the software. For this reason and with the help of external consultants the SSC have established a customized UML methodology consistent with the used standards and the maintenance and modification processes. Other UML methodologies are for example the Telelogic Harmony-SE, INCOSE Object-Oriented Systems Engineering Method and IBM Rational Process for Systems Engineering (RUP) (see [8]).

After that, the consultants have trainend a group of SSC engineers that work currently in several software projects. The most of them had no experience with UML before.

The training comprises of four periods: software use case modelling, software domain analysis modelling, software architectural design modelling and software detailed design modelling. These parts represent the steps of the left side of the "V" in the V-Model. After each period a coaching for the current software project was organized.

5 Conclusion

This article describes a new UML tool-based methodology to create the software requirements analysis. The methodology is use case driven and uses natural language requirements as well as UML diagrams. The methodology was established specifically for the SSC and should be applied in future for the software modification and maintenance of helicopters NH90 and Tiger. External consultants have trained SSC employees in both UML and UML methodology. The article shows how the UML model is build step by step with the aim of generating the software requirements specification. The later stages of the development process are not covered in this paper. But the UML methodology for the software design was already established.

Acknowledgement

The authors would like to thank all colleagues in the SSC for their feedbacks and advices. Especially, the two consultants of the SOPHIST GmbH for helping us to establish the described UML methodology and for their excellent training.

References

[1] *CMMI for Development, Version 1.2 – Improving processes for better products*. Carnegie Mellon University, 2006.

[2] TOPCASED Gendoc2 v1.5.0 tutorial, 2011. available at: http://www.topcased.org/ (visited 2011-12-01).

[3] *IEEE Recommended Practice for Software Requirements Specifications*, IEEE-830-1998.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[5] A. Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, Sep-Oct 1997.

[6] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[7] DO-178B. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[8] J. A. Estefan. Survey of model-based systems engineering (mbse) methodologies. *Jet Propulsion*, 25:1–70, 2008.

[9] R. Faudou, T. Faure, S. Gabel, and C. Mertz. Topcased requirement: a model driven, open-source and generic solution to manage requirement traceability. In *European Congress Embedded Real Time Software (ERTS)*, Toulouse, France, 2010.

[10] T. Faure, A. Haugommard, and J. F. Rolland. Gendoc2, generating ODT and DOCX documents from EMF models with TOPCASED. In *First TOPCASED Days*, Toulouse, France, Februar 2011.

[11] P. Gast and O. Bender. EADS GUIDELINE – using UML for software analysis and design. *EADS internal paper*, 2009.

[12] M. Glinz. On non-functional requirements. *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26, 2007.

[13] J. Gorman. Use cases – an introduction, 2006. available at: www.parlezuml.com (visited 2011-11-07).

[14] J. Killguss. Prozessverbesserung UML. *SSC internal paper*, 2010.

[15] Object Management Group. *Unified Modeling Language: Superstructure Version 2.3*. Number formal/2010-05-05. May 2010.

[16] D. Pilone. *UML 2.0 in a Nutshell*. O'Reilly, 2. edition, 2005.

[17] C. Rupp and die SOPHISTen. *Requirements-Engineering und –Management*. Hanser Verlag, 5. edition, 2009.

[18] C. Rupp, S. Queins, and B. Zengler. *UML 2 glasklar*. Hanser Verlag, 3. edition, 2007.

[19] R. Walter. Flight management systems. In *Cary R. Spitzer: The avionics handbook*, chapter 15. CRC Press LLC, 2001.

Author's Biography

Thomas Weyrath is project manager in the department for Integrated Systems in the Business Area Aviation at ESG. Since 2009, he has been working in the SSC and manages the process improvement project UML. Previously, he worked in the Automotive Area at ESG as software engineer and project manager for 9 years.

Berthold Schinnerl is army aviation officer and systems engineer in SSC. He worked as development engineer for the TIGER. Beside systems engineering he received his PhD in Electrical Drives and Actuators in 2009 awarded with two research prizes in 2007 and 2009.

Franz Schöttl was working on the aviation system development of the Tiger helicopter between 1990 and 1996. Between 2000 and 2004 he was working on specifying the ASAAC Standard for future aviation projects. Since 2009 he has been working on the process improvement project UML within the SSC.

Herbert Schreyer works on the aviation system development of the Tiger and NH90 helicopters since 1995. 2005 he joined the SSC where he took over the role of a system architect. Since several years he is charge of evaluation and introduction of new methods of avionic definition.