AN OBJECT-ORIENTED METHODOLOGY FOR MANAGING THE COMPLEXITY OF ATM SYSTEMS

Michel LEMOINE and Pierre DARBON
ONERA Centre de Toulouse
2 avenue E. Belin
31055 Toulouse CEDEX
Michel.Lemoine@onera.fr

Keywords: ***object-oriented approach, static and dynamic descriptions, validation, DSS, CDM, ATM systems***

Abstract

In this paper we emphasize how a computer science technology, namely the Object-Oriented (OO) approach, is able to help modeling and (in)validating large and complex systems such as ATM systems, airport ATC…

Such models are the right basis for a deep analysis of properties that current or future systems should have. Systems can be modeled from two points of view: static and dynamic.

We will focus our presentation on the methodological aspects. First of all we have a glance on our methodology, and the used notations. We then will exhibit a small example extracted from a current large project conducted at ONERA.

Finally we will show how our models are a firm basis from which fast prototypes can be easily derived. These prototypes are the first development steps for Decision Support Systems (DSS) that can be developed and implemented according to some strong requirements.

1 An OO methodology for mastering the complexity of large and complex systems

Our methodology is largely derived from the Fusion method [1] that is an OO method using the UML [2] notations.

In our methodology, we apply a general System Engineering process as this one recommended by the standard IEEE 1220 [3]. In other words we have adapted the Fusion method to this standard. We are interested by the static and dynamic properties the models we build have or have not.

In the following we introduce our methodology from the point of view of the development. But it may be applied in the same way for existing system.

1.1 Short review of the methodology

Our methodology takes care both of the above characteristics of ATM systems and of the fact that many ATM sub-systems do exist. Indeed we do not suggest a methodology for building only new systems, but rather a methodology for representing both current and future systems, regarded as improvement of existing systems. Here is the main keyword of our methodology: the ability of extending and improving existing systems!

Our methodology is composed of four main phases:

1. Representation, both from a static and dynamic point of view, of existing systems or of knowledge of the domain we are considering: the by-products are UML models.
   Static and dynamic models allow giving different and complementary points of view about the system under consideration. Indeed before any real development it is mandatory well understanding the system we are designing. This can be achieved by means of complementary models as those offered by the UML notations.
2. Validation of the UML models by real end-users.

After modeling a system, it is mandatory to validate it in order to guarantee it meets its requirements. This is done informally according to an inspection as suggested by [6]. For ATM systems end-users are Pilots, Air Traffic Controllers subsidiary of Civil Aviation Authorities, Airport Authorities...

3. Improvement of models, and again their revalidation by end-users.
   In our methodology each validation step is followed by an improvement step, which consists of redesigning the invalidated models. These redesigned models are again validated/invalidated until we arrive to a full validation by end-users.
4. Derivation of Decision Support System (DSS) and/or Collaborative Decision Making (CDM).
   In our methodology we are not considering a complete development process. We stop at the level where fast prototypes can be easily derived from improved and validated UML models. DSSs or CDMs we deliver are only fast prototypes that can be installed in real situation, but of course that are not as accurate as a final system.

This four-phase methodology is based on the Fusion method and UML notations. We review the four phases in the following.

1.2 More details on our OO methodology

1.2.1 Phase 1: modeling existing systems

Model building of existing systems or the knowledge we have about the domain derives directly from the Fusion method.

We first of all introduce Use Cases that are responsible for identifying: the actors of the system, its functionality, and its boundary. It is not our purpose to describe in details how the Use Cases are used, nevertheless it should be understood that using a strong methodology is the only way of success for large and complex systems. This first step has to be validated by end-users in order to guarantee that no important functionality and none actor have been forgotten. It is as well important to clearly identify the boundary of the system, i.e. what functionality belongs or not to the system.

2. We identify, from each Use Case, a set of dialogs or scenarios describing all the available interactions between the system under consideration and its actors[1]. The right identification of scenarios corresponds to designing a robust[2] system. Indeed scenarios must encompass both nominal and non-nominal actions.
3. We then translate some appropriate scenarios into Sequence Diagrams, one of the UML notations. This third step allows introducing some potential objects offering services. A service is nothing more than an operation, i.e. function provided by an object.
4. We derive, from all Sequence Diagrams, a Class Diagram. A Class Diagram represents the static view of a system. We only represent classes, seen as a set of objects with the same properties, and relationships between classes.
5. Finally from each class is built a State-Transition diagram that allows considering the dynamic behavior of any instance of the class. During this step (part of) the robustness of the system is established. Indeed non-nominal actions are reintroduced in the dynamic behavior of the class where these non-nominal actions take place.

1.2.2 Phase 2: validation of static and dynamic models

The second phase corresponds to a validation phase, validation meaning: *am I building the right system?* whereas verification means: *am I building the system right?* as proposed by W. Boehm in [7].

In other words, do the abstract models represent what we expect? Each time we produce a new set of information as described

[1] An actor is either an end user (a pilot for instance) or an external system with which the system is interacting (for instance the forecast).

[2] A system is considered as robust if, and only if, we can guarantee every event that may occur during its life time is considered and treated in such a way the system will never be destroyed.

above, end-users must be asked to validate them.

This is easier than with any other methodology. Indeed the UML notations are both: **simple** - the models presented below are, hopefully, convincing, and **meaningful** - to each used icon of UML notations corresponds a clear semantics.

Validation is mandatory for each kind of models. It allows not going on the development process when a failure is discovered. It must be noticed that we must trust end-users each time they claim there is an error. But when they claim there is no error, unfortunately we can't trust them. Thus the inspection approach leads more often to invalidation than to the guarantee that what has been designed is valid!

1.2.3 Phase 3: improvement of existing models

Having designed a set of application validated OO models, we are able to attack the next phase that will help improving existing systems.

This is possible because we can take advantage of the *easy-to-modify* object-oriented capability. In the OO approach, modifying the design consists mainly of modifying and/or updating the Class Diagram. Indeed errors discovered by end-users correspond most of time either to misinterpretation by the designers, or discovering that some information, date, etc. are missing.

For the former, updating the different models is quite easy since most of the supporting tools allow doing it in a very simple way. Moreover, since the main interest of the OO approach is to give responsibility to each object, any update is local to a few objects, very often one and its relationships with the others.

For the latter, again updating the different models is quite easy since in general adding new information consists of adding either new classes, or adding new attributes/operations to a class, or adding new relationships between existing classes. In the latter each update does not disturb diagrams since information is added, without changing others.

In both cases we may add/remove information without difficulty while preserving almost existing and secure properties of the modeled system. We present in the following a rather convincing example.

1.2.4 Phase 4: designing and prototyping of DSSs and CDMs

This final phase is a direct consequence of the preceding one.

As soon as some improved models of an existing system are available, we follow the Fusion method in order to transform some of its parts into either a fast prototype or an operational system.

The latter is the classical way of developing software with the OO approach. In our methodology we recommend to set up this step only when the former has been applied.

The former, which corresponds to a prototyping step, has as main result a software tool that can be carefully tested and checked by the concerned end-users.

In the example presented below we have restricted ourselves to fast prototyping of a DSS.

2 Application of our methodology to an ATC subsystem

2.1 An ATC airport subsystem: a few characteristics

ATM and ATC systems have, among others, the following characteristics:

- They are critical: they must respect very hard safety constraints imposed by the aviation authorities.
- They are real-time: they must respect strong time constraints.
- They are inherently complex: the amount of available information is very important. They must take into account different kind of information coming from different sources:
    - Information provided by the radar system and other sensors: magnetic loops for instance.
    - Information pertinent to airlines: payload, which corresponds to a weight of aircraft, number of passengers.

- Information from the regulation authorities: the Central Flow Management Unit (CFMU) for all the European flights gives the available times.
- The weather conditions play a significant role. They may evolve quite frequently.

New procedures are very often imposed:

- By airline companies both from an airside and landside point of view.
- By ATC authorities in order to improve safety or to better control the environmental impact.
- By aircraft constructors: new technologies such as D-GPS, Data Link, are directly implemented in aircraft.

2.2 An example based on an airport ATC subsystem: the AdF project

In order to convince the reader of the usefulness of our methodology, here is a small example borrowed from the *Aéroport du Futur* project [4] and [5].

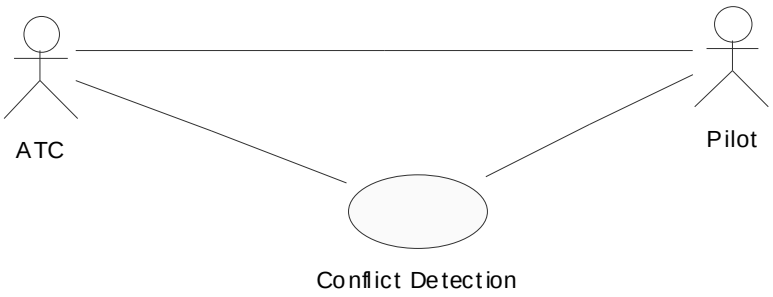

Figure 1: Use Case of a DSS monitoring the aircraft movement

2.2.1 Building a DSS able to identify conflicts between taxiing aircraft

When aircraft run on taxiways many problems may occur. Thus the ground air traffic controller who is standing in the tower has the responsibility of taking care of the aircraft traffic. In other words he/she must consider the situation – aircraft running on taxiways – and send them the right orders. He/she can stop or move aircraft, depending on their relative distance whenever he/she is convinced there is a potential danger.

We model the situation from a functional point of view as shown in Figure 1. We have identified:

1. Two actors: the Pilot and the Air Traffic Controller (ATC).
2. One main functionality: the Conflict Detection Use Case that will become a DSS.
3. Three kinds of communication: one between the Pilot and the ATC, one between the Pilot and the Conflict Detection system, one between the ATC and the Conflict Detection system.

The communication between the two actors corresponds to the current situation. It is no longer part of our Conflict Detection system.

The communications between any actor and the Conflict Detection system are the only ones we will consider because they correspond to all the possible interactions between the Conflict Detection system and the two actors.

2.2.2 A simple take-off scenario

The interaction between an actor and a Use Case must be described by informal scenarios, and then by semi formal sequence diagrams.

In order to be able to design a robust system, we have suggested considering two kinds of scenarios: those that correspond to nominal situations, and those that correspond to non-nominal situations.

It is of first importance to identify clearly theses two kinds of scenario, and to show that they partition the space of possible scenarios. This is the only way to guarantee that the system to be designed is robust.

The following scenario describes a conflicting scenario. It has been validated by the ATC of the Blagnac Airport - Toulouse.

Informally when an plane (here the object ABC618, instance of Plane class) will leave its gate it must obey the following scenario, which is subdivided into a non conflicting one, followed by a conflicting one.

1. It[3] asks for a start-up and departure clearance before starting its engines.
2. This clearance is given by the object John, a ground air traffic controller, who knows the awaken flight plan[4] of ABC618.
3. Then the object ABC618 asks for an approval for the push back procedure.
4. John answers positively the plane, which then transmits the push back approval to the tow manager.
5. John asks the planning management (it can be either an automatic system or John himself) to add ABC618 into its planning.
6. A taxi clearance is given to the plane, allowing it to taxi.
7. Now John activates the ABC618 flight plan in order to inform the Flight Plan Processing System of the airborne time.

Here ends the nominal scenario for leaving the gate and taxiing. We continue the scenario on the taxiway.

8. A sensor i.e. the surface radar monitors all the planes on the platform.
9. It informs as well a conflict detection system (which in classical ATC is the John's responsibility) of all the events - mobile movements … - happening on the platform.
10. Another object, plane DEF233, is detected by the sensor.
11. The conflict detection system is able to detect a potential conflict between ABC618 and DEF233.
12. John, informed by the conflict detection system of this potential conflict, asks the object DEF233 to stop taxi.
13. Once the conflict is solved, John asks the DEF233 to resume taxi.
14. And so on.

This scenario is then translated into a sequence diagram (see Figure 2) on which we identify objects, and their communications. In OO terminology, objects exchange messages. In any communication there are two objects: the emitter, and the receiver.

Here we have identified as objects:

- planes ABC618 and DEF233,
- some sensor (for instance a surface radar),
- the flight plan of ABC618,
- the planning which is a tool that helps the ground ATC,
- the ground ATC who represents one of the actor,
- the tow manager whose responsibility is to initiate some operations,
- and finally the DSS, another tool that detects conflicts.

It must be noticed that we have decided just to design a DSS. But as it can be seen from these scenarios and the corresponding sequence diagram, it can be easily extended to a CDM system, i.e. a conflict detection system able to take some right decisions.

Having validated the sequence diagrams (we have been obliged to consider all the other known conflicts), we can move to the next step: the construction of the Class Diagram.

[3] *It* represents the plane!

[4] The awaken flight plan corresponds to the initial data of the flight plan.

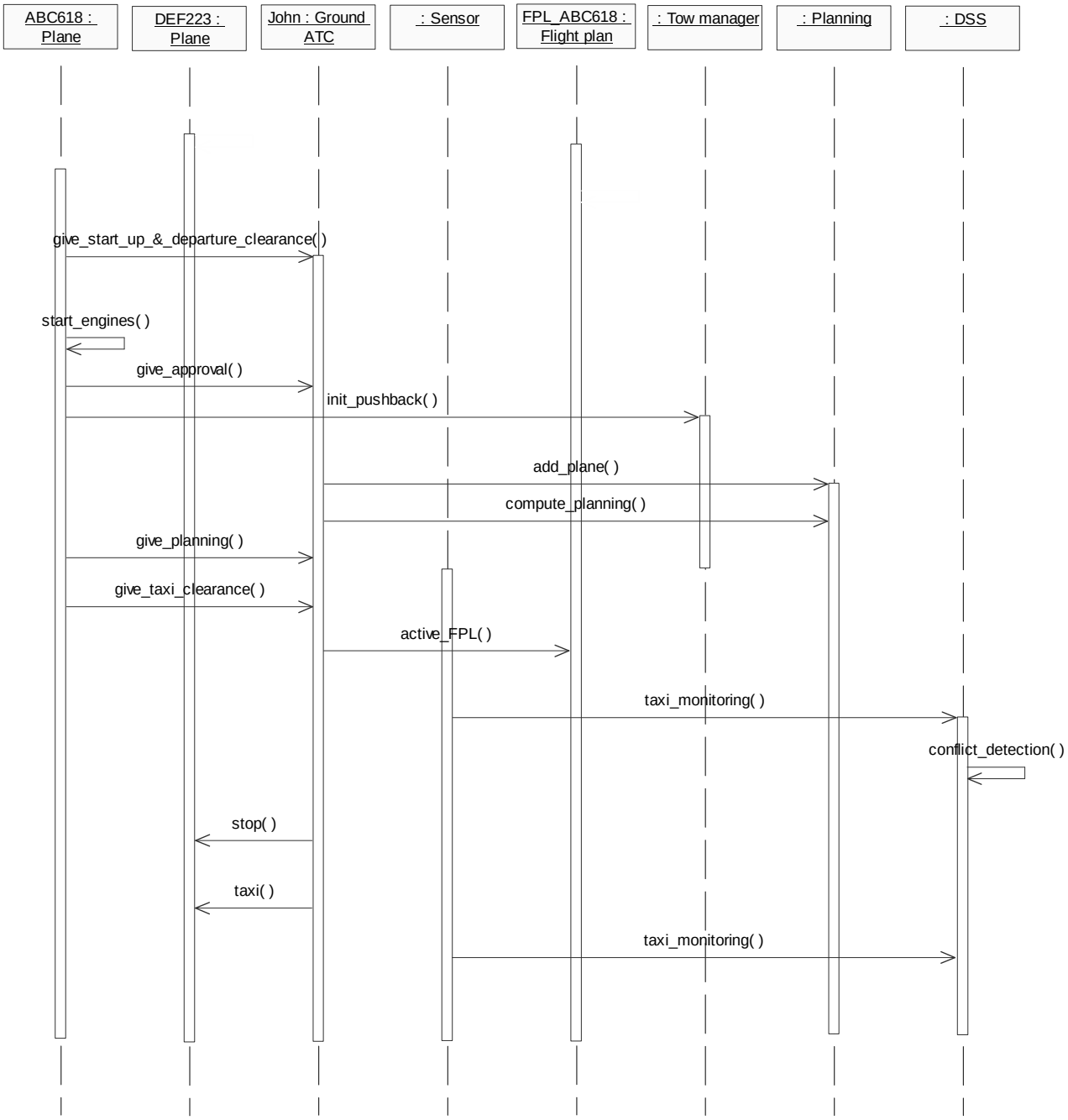

Figure 2: The sequence diagram associated to the scenario of §2.2.2

2.2.3 The Conflict Detection Class Diagram

The Class Diagram (see Figure 3) is part of the *Aéroport du Futur* Class Diagram. We have restricted it to the set of classes that appear in the above scenario.

Any class i.e. the set of objects it represents, has attributes or internal data that are not shown here. It offers services that other objects can require as soon as they are in its scope. Moreover there are relationships between classes, which represent strong links that do exist at run time. For instance the object Tow manager is linked with an object (here one and only one instance of a Plane) with which it is able to exchange information. The duration of links depends on the context. For the Tow manager and the Plane, this link is limited to the push back procedure.

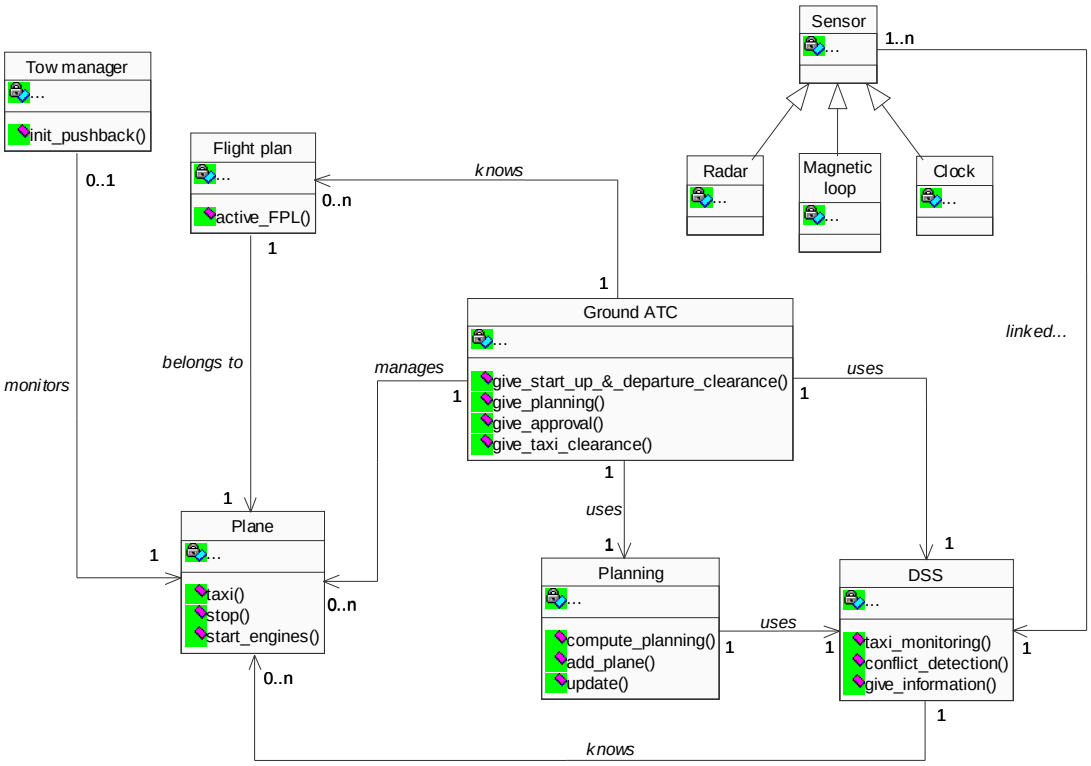

Figure 3: Class Diagram

It must be noticed that a Class Diagram does not show how the services offered by Classes are used. This is represented in other Diagrams such as State Diagrams and Activity Diagrams. Here the relationships translate the fact that for a while an object such as Tow manager is in Relation with an Object such as Plane. From the Class Diagram point of view, an object Tow manager is created each time a plane pushes from its gate. It dies as soon as its activity is off. On the other hand any plane is in relationship with 0 or 1 object Tow manager. And so on.

One main interest of such a Class Diagram is that only static information is modeled. Moreover it is obvious that we have reused without changing them other classes still existing in other Class Diagrams. This is clearly part of the interest of the OO approach. We are designing a new system – here the Conflict Detection system – based on previous class diagrams that model other part of the airport air traffic control.

2.2.4 Dynamic aspects: a state-transition diagram

Having modeled static aspects, it is time to model behavioral aspects. For this purpose we describe state-transition diagrams, their notations having been borrowed from D. Harel [8].

In a state-transition diagram we described the complete behavior that all the class instances follow. For instance, let us considering the behavior of a plane as this one represented in Figure 4.

Starting from an initial state, the plane arrives in an intermediate state (Interm1) where it does wait for the first clearance it has asked. It must be noticed that even the emitter is the plane, the corresponding event is not part of the plane state-transition diagram but part of the ground ATC sate-transition diagram. Indeed in a state-transition diagram, we represent only the events received by the instance of the class under consideration. Thus being in the state Interm1, the plane is waiting for the clearance

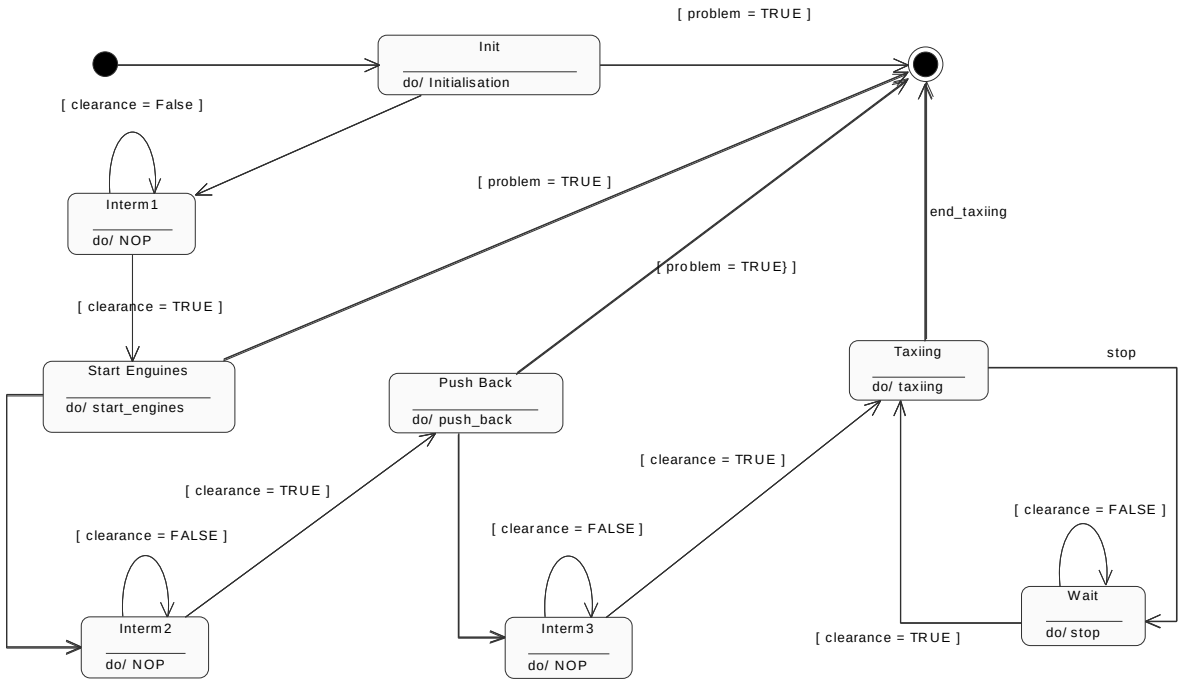

Figure 4: the state diagram associated to any plane

corresponding to the service *give_start_up_&_departure_clearance* offered by the ground ATC.

As soon as the answer is positive, the plane state becomes Start Engines in which it can start its engines.

Then it moves to the Interm2 state in which it is waiting for the push back clearance. When it arrives it moves in the Interme3 state in which it is waiting for the taxiing clearance. When it arrives the plane state becomes Taxiing in which the plane is running on taxiways.

During taxiing the plane can either be stopped or arriving to its final state. In the former it will take off, and its state becomes undefined. In the latter its state becomes Wait until a new clearance is given. As soon as the clearance arrives the plane restarts taxiing. And so on!

It must be noticed that at any stage a few problems may arrived. Some of them have been stored as *[problem = TRUE]*. In other words any time a problem arrives the state plan becomes undefined.

2.2.5 Building a prototype for the Conflict Detection system

As described in our methodology the next step consists of deriving fast prototypes from the designed models.

Part of the work is easy. Indeed most of the commercial OO tools do support an automatic translation of the Class Diagram. In other words we can derive automatically an implementation of the set of classes that constitue the Class Diagram.

This translation is not complete since we have not provided the operation semantics in our models. We do that by translating the informal semantics expressed in the *do/ statement* that appear in each state-transition diagram. It must remeber that to each class is attached a State-Transition Diagram that describes the global behaviour of each instance of the class. In the *do/statement* we just express informally what is performed when arriving in the state.

The language which seems the most suitable for fast prototyping is Java [9]. It is

very simple to use. It supports all the safe principals of OO. Moreover Java is platform independant.

The main interest of any prototype is to give access to another kind of validation. Indeed end-users such pilots, ground ATC, airport authorities, etc. are not familiar enough with UML notations. Even it is possible to educate some of them, the best way to validate a system remains in using an operational model of the system to be developed.

As far as the fast prototype performances are good enough, and as far as it is a realistic approximation, it is obvious that end-users may interact with it and validate it.

3. Conclusions

Our methodology, briefly sketched above, allows first of all managing the complexity of large systems. In the *Aéroport du Futur* project more than 200 classes and relationships have been identified.

A second important point is that end-users can both informally and formally validate all the developed models, restricted to the application level. In the case of the *Aéroport du Futur* project, all the diagrams and their contents have been validated by end-users, mainly ATC and pilots.

A third point is relative to the easiness we can improve existing models in order to take into account for instance new procedures. In the above we have suggested how introducing one simple improvement about a Conflict Detection system an ATC was aware of. He established indeed the requirements. Of course he also explained how the conflicts are detected, and what actions must be taken when such a situation arises. It must be noticed that conflict detection is depending on the ATC who is in charge of the air traffic management. Indeed the notion of conflict is not formally set up by the air traffic regulations. Consequently, each time we, as computer scientists or software engineers, design a DSS or a CDM system we formally described a situation that is in no way formally described in books or by procedures. In other words we are faced with some problems that are not necessarily considered in the same way by other ATCs. Their validation is then mandatory!

The example given above has been developed in a few weeks. It was formally validated by a set of ATCs. It then was transformed into a Java application, taking advantage of still existing Java classes.

The last and important advantage of our methodology is that having developed a first prototype, as above in Java, it can be easily transformed into a *Federe* that is easily integrated into a large HLA [10] *Federation*. It is important to consider we have to develop large simulations, integrating many small more or less independent simulators, i.e. fast prototypes, as suggested in [5].

References

[1] Coleman D. et al. *The Fusion Method*. Object-Oriented Development. Prentice Hall International. 1994

[2] Booch G., Rumbaugh J., Jacobson I. *The Unified Modeling Language User Guide*. Object Technology Series, Addison-Wesley, 1999

[3] *Standard for the Application and Management of the Systems Engineering Process*. IEEE 1220-1998

[4] Lemoine M. *Managing the Airport Complexity: Role and Design of an Information System*. 22nd ICAS Congress (International Council of the Aeronautical Sciences), Harrogate, UK, August 2000

[5] Adelantado M. *Experimenting the HLA framework for the ONERA project "Airport of the Future"*. Fall Simulation Interoperability, SISO 1999, Workshop, Orlando, USA, 1999

[6] Fagan M. E. *Design and Code Inspection to Reduce Errors in Program Development*. I.B.M. Systems Journal, Vol 15, n° 3, 1976

[7] Boehm W. *Software Engineering*. IEEE Transactions on Computers, Vol. 25, p. 1226-1241, December, 1976

[8] Harel D. *Statecharts: A Visual Formalism for Complex Systems*. Sci. Comput. Programming, 1987

[9] Flanagan D. *Java in a Nutshell, 4th Edition*. O'Reilly, 2002

[10] *High Level Architecture*, IEEE Standard P 1516, 2000