Athéna: a generic multi-purpose environment for simulating complex systems

Jean-François Tilman
Philippe Morignot
Jean-Clair Poncet
Nelly Strady
Axlog ingénierie
19-21 rue du 8 mai 1945
94110 Arcueil, France
(+33) 1 41 24 31 00
{Firstname.Lastname}@axlog.fr

Bruno Patin
Dassault Aviation
78 quai Marcel Dassault
92552 Saint-Cloud, France
(+33) 1 47 11 58 54
Bruno.Patin@dassault-aviation.fr

Keywords:
Generic simulation framework, distributed simulator, multi-agent, HLA.

ABSTRACT: *In this paper, we present a generic multi-purpose simulation environment, called Athéna, which has been developed to simulate the behaviour of systems in complex environments. This simulation environment allows to merge discrete and continuous aspects by using discrete automata on the one hand, and continuous parameters and interactions on the other hand. These aspects can be tailored by the user through shared libraries and a scenario description language. Simulations can be distributed in a client/server mode through a network, to take advantage of heterogeneous workstations. It can also be interoperated with other simulations through the High Level Architecture standard. Current applications of Athéna include: (1) simulation of a package of autonomous unmanned aircraft in missions such as attacking targets in hostile territories; (2) embedded system architectures described with AADL (Architecture Analysis & Design Language).*

1. Introduction

Simulation is often a cheap and fast alternative to test systems under development. It enables the representation of their behaviour in a simulated environment and delay the moment when they will be really integrated in the complete platform. This is even more true when working in research and development domains, where the target platform hosting the developed system does not exist yet. Simulation also enables to adapt parameters, experiment many configurations, modify the environment, and then stress the tested system to evaluate its performances.

Many simulators exist, which can for example be used in avionics or defense domains. Some of them, like Escadre [1] are able to support a large scale of applications, but they require strong programming skills. Other ones, like the Stage simulator family [2], are more intuitive but are also dedicated to more precise industrial problems (aircraft or helicopter simulation). Nevertheless they are not well adapted to projects out of their scope. A simulator like RT-Sim [3] is able to simulate the execution of any user code, and the configuration of the simulated architecture can be easily done through a user interface, but such a simulator is too low-level for some purposes, since it is dedicated to the simulation of the embedded real-time code itself. To cover multi-agent domains some simulators also exist ([4, 5]), but they are often too specialized and don't support simulation of the environment. Thus, it is difficult to find a simulation environment both easy to use by a non-programmer user, and generic/flexible enough to be adapted to research subjects.

In this article we present Athéna, which has been developed to cover this lack. Athéna is a generic simulation environment used in unmanned aerial vehicle (UAV) research area, but also in other domains like space or real-time system applications. We explain its concepts, its extensions and

genericity capabilities, and the distribution and interoperability aspects. We also present two examples illustrating the interest of Athéna in relation to other solutions: To support the development of embedded autonomous functions for UAVs, and to build a simulator for the simulation of real-time system architectures.

2. Overview of Athéna

2.1 Raison d'Être of Athéna

Athéna [6, 7] was born in 1998 from the need of Dassault Aviation to develop advanced software for Unmanned Aerial Vehicles (UAVs). This includes, among others, prototyping autonomy functions for mission management and assessing their efficiency [8]. Due to the criticality of such a system, real aircraft prototyping can not be foreseen and simulation is the only way to test the autonomous aircraft behaviour. In order for the simulation to be useful, it has to be fairly realistic: the target simulator has to be able to describe complexe environments with the sufficient level of details and to focus on specific aspects if needed. A lot of simulation tools are available, many of them fully adapted to the avionics domain and supplying accurate simulation facilities. So, why did not we choose one of them?

Firstly, for the UAV simulation, all the interesting matters come from the autonomy functions and specific high-level data used as inputs and outputs. So, most of the services provided by the COTS simulators (e.g., Escadre [1], Stage [2]) are not useful for us. A crucial issue in our problem is to be able to integrate user-defined elements, behaviours and processings in the simulation. An alternative could have been to use a generic agent simulation environment: encapsulating the autonomy functions in agents, it would have been possible to simulate it in a parameterizable environment. The main problem is that the "generic" simulation environments are generally speaking not so generic: for instance, some environments are really efficient to model a specialized domain (fire-fighting in the case of Phoenix [4]), but are difficult to adapt to other situations where there is no discretized space with data on cells. Moreover, even in the case where they are generic enough, they focus on the multiple-agent system simulation, with less capabilities on the environment description. This is for instance the case of the ARCHON system [5], in which the agent concepts are well suited for advanced functionalities, but in which the environment description and the monitoring module are too poor for us.

Secondly, an important point of the simulation is to be able to assess the tested component efficiency in different heterogeneous situations, with the ability to focus on several particular points and refine specific environment models in parallel with the tested component refinements. This results in the need of a modular, easy to parameterize, simulation: we want a means to quickly prototype short scenarios, only using basic components such as data and functions to handle them as well as precise detailed ones with well-described components and realistic behaviours. This means that we have to be able to simulate simple components, which are able to implement the minima of the needed airplane system behaviours, as well as to replace or change them as many times as possible with minimal cost, according to new developments or previous simulation results. It results that we do not have the need for all the high level functions provided by the commercial simulators, while we have requirements for a high genericity and scalability level of simulation core software.

In parallel, some signal processing functions are used by Dassault Aviation for particular avionics domains such as Radar Cross Section processing. The simulation environment has to enable the execution of these functions, for instance, to model the realistic behaviour of a radar. Thus, the target simulator has to provide a means to easily introduce elementary user's functions and to compose them, in order to plug the resulting functions in the final simulation environment. The general underlying idea is to be able to replace step by step the basic components designed at the beginning of the simulation by the real signal processing ones whenever we can. This continuously refines simulation accuracy and improves precision of results.

Moreover, Dassault Aviation expressed the need for its simulation environment to be usable by operational experts, for example, and not by computer scientists. Most of the available COTS simulation environments need strong programming skills (e.g., Escadre [1]), which is not acceptable for us. Hence the decision to design a specific simple high-level language to describe a simulation. For modularity purposes, this language introduces a few simplified object concepts, to enable composition and limited inheritance. From this point of view, the Athéna language can be compared to SMILE [9], but with a somehow more intuitive syntax. This way, two kinds of users have been defined for Athéna: the basic user, not necessarily a computer scientist, who is able to describe a simulation using available libraries, and the advanced user, a computer scientist, who will write libraries of objects which are usable in the simulation environment.

Finally, Athéna has to fulfill different classes of requirements:

- Design a flexible simulation environment for autonomous functions. This is done through the use of elements extracted from libraries and composable

using a simulation description script;

- Allow for composition of processing functions, which are called them during a simulation. It must be pointed out that such an integration leads to the fact that we do not have a real-time simulation. No other mechanisms than the computing power are used to ensure such a behaviour;

- Enable to distribute the simulation over a network of workstations to perform efficient simulation of processor consuming processing and to allow private components to work with public ones in compliance with property rights or confidentiality requirements.

The need to introduce some vehicle simulation elements for UAVs and some processing simulation elements for signal processing has required to design a system that can handle both event- and continuous-based simulation elements. These requirements also result in a design oriented through easy integration of user's functions. This requires to provide some data types (used for signal processing as for other simulation purposes) and the basic tools to use them. Afterwards, considering all the requirements, it appeares that the envisaged design is not specific to avionics domain but turns out to be generic for a large range of applications.

3. Athéna Description

3.1 Global Architecture

Like most distributed applications, Athéna is made of interacting servers and clients. On the one hand, each server hosts part of the simulation objects, as parametrized by the user, and is in charge of managing the simulation execution. The different servers are in charge of co-ordinating themselves to have a coherent time unfolding and to activate the different simulation components. This is enabled by a distributed synchronizer and an object manager. The object manager is able to create, contain and destroy the simulation elements. It is also used to start and stop the execution of the simulation. The distributed synchronizers are contained by the servers (one per server) and converse through the network to maintain a consistent logical date for the whole simulation. The servers also load the list of libraries containing the implementation of the different objects and functions to be simulated. On the other hand, a set of clients can be deployed on the network: one for the simulation initialization and control of execution, and others for visualization, traces, data storage, etc. Except for the simulation controller, the other clients are optional and not directly related to the simulated functions. Interfaces are provided to the advanced-user to enable the development of additional specific-purpose clients.

The distribution issue in Athéna relies on the use of a CORBA (Common Object Request Broker Architecture) bus. CORBA is an international standard that provides a way to use objects over a network just as if they were local [10]. In Athéna, all the simulation elements like synchronizers and object managers in the servers are CORBA objects. This means that they can receive requests from any other element, or from a client application, and answer these requests according to predefined remotely accessible methods. A CORBA naming service is used to collect references to all the elements and establish the communications among the simulation objects. Once all the objects have been registered to the naming service, the different clients and servers are able to use CORBA facilities to communicate (Figure 3.1).

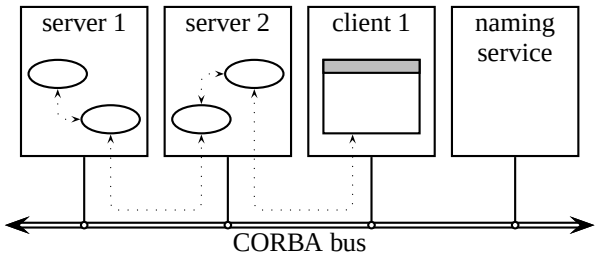

Figure 3.1: General architecture of Athéna

Two different notions are important to understand the behaviour of the simulation environment. The first one, *distributed elements*, is attached to the distribution aspect. A distributed element is a simulation object for which the user is able to specify the target server containing it. A distributed element is accessible from any other simulation component through a CORBA reference. A data container, such as the position of a robot, is a distributed element.

The second notion is the one of *sequenced object* and deals with the simulation unfolding aspect. A sequenced object is a particular distributed element that is periodically activated by the server managing it to perform actions. The term *sequenced object* comes from the notion of sequencement of activations in the servers at the different simulation time steps.

These two notions are the basis for the components described in the following sections.

3.2 Basic Standard Components

One of the objectives of Athéna is to enable building simulations by assembling low level components such as data containers or handling functions specified by the user. The domain-specific high level components (aircraft, vehicles, radars, etc.) can be built using these basic components.

They can also be put together in user libraries. But these components are not provided by the simulator itself.

To simulate a continuous domain, as needed by a model of some physical world aspects (dimensions, temperatures, etc.), two kinds of basic components exist: the *parameters* and the *interactions*. A *parameter* is a data container which may be read or modified during the simulation. Some special parameters are already defined for the usual data types: integers, floats, strings. New ones can be introduced to handle more complicated data, provided that they respect interfaces of the simulation environment. An *interaction* is an element which periodically executes a user function. It is the main user of the parameters. Conditions can be added to enable or disable the activation of interactions depending on various criteria: period, state, condition on specific parameters.

It is also possible to model discrete aspects such as automata using provided basic components. The *states* are used in conjunction with the *transitions* to represent various states of the simulated system and the modification of these states. These states can be used to enable or disable the activation of other elements such as interactions. *Events* are other discrete elements. They are used to change states, by firing transitions, or to enable and disable the activations of some elements. They depend on various conditions, which can be based on the value of parameters, states or other events.

When creating a scenario, the user must be able to prepare complex components, and then use several copies of them. This is done through the notion of *instance*. An instance is a generic container that can include parameters, automata and interactions. This notion is very close to the object notion in object-oriented programming. For example, we may need to model an aircraft and use this model to compose a formation. In this example, if the aircraft model contains a parameter *position*, then each instance of this model will have its own instance of the position parameter.

The first projects using Athéna have shown that several higher level components are often used. So, it became interesting to supply them with Athéna to avoid their redefinition in each project.

When modeling an aircraft, the user may need to model embedded real-time tasks and how they are scheduled. The user then provides two new components. A *task* represents a real-time task and specifies its attributes (period, computation time, priority, deadline). A *process* represents a processor containing tasks and activating them with respect to a particular scheduling policy (first-in first-out, rate monotonic, deadline monotonic, highest priority first, earliest deadline first).

Interactions and *tasks* are useful to call one user function during the execution of the simulation. However, this mechanism can be too limited in some cases, when the user needs to combine many functions to produce the expected result. A *processing* element is provided to solve this problem. A processing is either a *simple processing*, which calls a user function, or a *composite processing* which recursively calls other processings. The composition can use conditional structures and loops (i.e., if, for, while) to behave differently depending on external conditions or internal results.

3.3 Scenario Description Language

The creation of a simulation scenario consists first in the composition of many elements to model the simulated system. Graphical clients are useful to help the user in this job, by providing him with a user-friendly interface. Once the scenario is ready, these clients can control the simulation servers to instanciate the simulation and execute it. Nevertheless, when we want to use the simulator in batch mode, or to keep scenario descriptions and constitute reusable component model databases, we need a means to record the descriptions in script files. This is why we have defined a specific description language. This language is used to specify the different aspects of the simulation scenario: simulation time step, simulation objects, simulation data. Figure 3.2 illustrates the use of this language to describe a very short scenario with four parameters representing the position and speed of a rover, and an interaction modifying them to periodically compute the new position with respect to the speed. In this example, no distance or speed unit is mentioned. Likewise, the duration of the time step is not given in the description. All this is only known by the user, and taken into account in the function *computeNewPosition*.

```
PARAMETER Long positionX = 0;
PARAMETER Long positionY = 0;
PARAMETER Long speedX = 10;
PARAMETER Long speedY = 5;

INTERACTION newposition:
  computeNewPosition(
    positionX, positionY, speedX, speedY);
END;
```

Figure 3.2: Short example of scenario description

A complete set of simulation description scripts can be used to specify a simulation scenario. The object-oriented script language supports inheritance and encapsulation. It allows for file references, enabling to decompose the simu-

lation into elementary objects. This makes clearer the simulation scenario description while maximizing the reuse of the different simulation elements.

The description langage also introduces a new concept, which does not appear among the simulation elements: a *prototype*. It can be compared to a class, that is the description of an abstract component, which will be used later to create *instances*. It supports the inheritance mechanism to simplify the creation of complex prototypes by deriving other ones. Of course, a prototype can contain anything an instance can. Figure 3.3 shows an example of a scenario description using a prototype. In this example we also use a new parameter type defined by the user: Rectangle.

```
PROTOTYPE Rover
  PARAMETER Long positionX = 0;
  PARAMETER Long positionY = 0;
  PARAMETER Long speedX = 10;
  PARAMETER Long speedY = 5;

  INTERACTION newposition:
    computeNewPosition(
      positionX, positionY,
      speedX, speedY);
END;

INSTANCE Rover myrover;
PARAMETER Rectangle limits =
  "-100,-100,100,100";

INTERACTION reachLimits:
  checkSpeedVector(
    myrover.positionX, myrover.positionY,
    myrover.speedX, myrover.speedY, limits);
END;
```

Figure 3.3: Scenario description using a prototype

Figure 3.4 shows an example containing discrete elements: two states *on* and *off*, transitions between these two states, events to trigger off these transitions, and use of a state as a condition to activate an interaction.

```
PROTOTYPE AircraftSystem
  PARAMETER DoubleTriplet position;
  PARAMETER DoubleTriplet speed;

  STATESET status Off, On = Off;
  EVENT putSystemOn @launchSimu = 1;
  EVENT cut @becomesNotAvailable = 1;
  TRANSITION Off :  putSystemOn -> On;
  TRANSITION On :  cut -> Off;

  INTERACTION RadarDetection :
    PERIOD 2,
    WHEN (On),
    DetectFunction(position, speed);
END;
```

Figure 3.4: Scenario description using discrete elements

4. Extension and Genericity Mechanisms

4.1 User Functions

One of the main specific aspects of Athéna is the fact that all the advanced computations are provided by the user. The mechanism we provide to enable this is based on the use of shared libraries. The user develops his functions with respect to a predefined prototype and encapsulates them into a shared library. At execution time, the library is dynamically loaded and the functions can be found thanks to their symbols. The description of the scenario is read to know the name of the functions which must be called during the execution.

This mechanism is one of the main reasons of the high extensibility and genericity of Athéna, since any source code can be included into a function. Because the development of these functions requires programming skills, only advanced users will be in charge of this development.

In some cases, generic helper functions are provided to allow the user to develop his functions using widespread toolsets such as Matlab, Scilab, GNU-Prolog, etc.

4.2 User Datatypes

The second aspect of the extensibility mechanisms is the possibility to introduce and manipulate user data types. The servers contain and manage generic CORBA objects, the *parameters*, without any particular semantics. So, they don't need to know anything about the contents of the data type elements, so long as they are parameters. Athéna already supplies support for common basic data types such as integers, floats or strings. But this is often not enough.

New parameters can be developed (e.g., complex numbers). They can contain any data provided that they respect requirements coming from the generic parameter class. These requirements include the implementation of function allowing reading and writing data.

The new parameter can enrich the generic parameter interface by introducing methods dedicated to the specific data it manipulates. For example, we can develop an array parameter with methods to access a given item, to sort the contents or to check the validity of the data.

4.3 Even more Extension Capabilities

Athéna, through a strongly layered architecture, enables lower level extensions mechanisms. By default, these ca-

pabilities are not proposed to any user and concern highly specific cases. Moreover, their implementation requires an advanced knowledge of Athéna structure and principles.

It is possible to create new CORBA objects extending the very basic elements which are handled by the server, that is the *distributed elements* and the *sequenced objects*. In this case, the server is only used as a container or a sequencer which will regularly activate the objects.

The last possibility to extend Athéna is the development of specialized servers. They will interact with the default server through the CORBA bus. This feature can be useful to establish a particular connection of Athéna to another tool.

Moreover, it is possible to use the CORBA interface of all the simulation elements to interact with them. Specific clients can be developed to control and visualize the simulation when the generic ones are not sufficient. This technique has been used in the development of ADeS (see 6.2).

5. Distributed Simulation

5.1 Running Athéna over a Network

Many reasons exist to distribute a simulation over a network. Of course the simulation of a huge scenario requires a lot of resources and may be improved by using several computers. In our case, the distribution also enables the use of a given platform to execute the tested function. This can be needed for instance for software licence purpose.

The distribution of servers and simulation objects over a network is done by using CORBA. The simulation managers (enclosed in Athéna servers) as well as the simulation elements are CORBA objects, remotely accessible. Naming conventions allow the different simulation elements to access each other thanks to the naming service. The control and visualization of the simulation are also done by clients using CORBA.

Using a widely-spread international standard as CORBA to support the communication process between the different elements of the simulation eases the development of specific clients and ensure the behaviour of the system. Moreover, this gives flexibility on the platforms on which it can be used and the language additional clients are developed with.

5.2 Interoperability with HLA

HLA (High Level Architecture)[11] is used by the simulation community to interoperate several simulations into a single *federation*, that is an assembly of *federates* collaborating together to produce the result. Athéna also supplies a support for HLA. This can be interesting to develop specific aspects of a simulation and to introduce them into a global scenario managed by another tool.

Several approaches can be imagined. First, let us suppose that we have another simulator. Then we use Athéna to provide it with the simulation of a very specific component, and use the HLA mechanisms to publish the results to the main simulator. In this case Athéna is the slave, the other simulator is the master.

Second, we can also use Athéna with other simulators at the same level. Each of them support an equivalent part of the scenario. This can be envisaged to put together the simulations of several vehicles provided by specific simulators into an environment. In this case all the simulators will collaborate together.

The flexibility of Athéna enables its connection with simulators having poor flexibility capabilities. Suppose that we have a simulator publishing data in a predefined format. It is possible to easily adapt the Athéna side of the simulation to handle the specific format of the data. This is done by introducing a specific user function, able to handle and transform the format of the data.

6. Examples of Simulations Using Athéna

Athéna has been already used for various purposes. We present here two examples illustrating the capabilities of this simulation framework. The first example belongs to the initial application domain of Athéna. The second one shows a completely different use of the simulation kernel to build a specialized simulation tool. Other examples are described in [12].

6.1 Mission Management System for Autonomous Aircraft

Managing Unmanned Combat Aircraft Vehicles (UCAVs), which have been studied for 25 years now, need some particular simulation facilities during the design and development phases. At design time, it is very useful to be able to simulate the behaviour of the envisaged system, while no system component has yet been developed. At development time, it is necessary to be able to test the developed system while they are iteratively prototyped. For both these aspects, the Athéna framework enables to simulate system behaviour at coarse grain, early in the life cycle, while continuously refining the simulation parameters to reach a full implemented realistic simulation environment at the end of the project.

The Mission Management System (MMS)[1] is an embedded system that manages in real time the different parameters of missions of UCAVs flying in package. A package of UCAVs can contain up to eight UCAVs flying together. Then, the MMS is a system distributed among all aircraft of the package that enables to manage the mission with very limited human intervention. Actually, human intervention is limited to emission of requests for information, emission of environment information and tactical orders (targets, threats, tactics). The main tasks of such a system are then:

- To determine the package path: to reach the target(s) and to come back according to pre-determined flight corridors[2];
- To determine the package configuration: to allocate a role in the package to each aircraft (e.g., communication relay, jammer, bomber) ensuring deconfliction of role allocation and resource usage;
- To plan the communication inside the package and between package and ground-based command and control;
- To take into account safety parameters (e.g., probability of survive, reaction to threats and tactics.

Of course, this list is not exhaustive. However, as we can see, the MMS has a large range of activities to perform, each one being corelated to other. The development of such a system takes advantage of a flexible and scalable simulation environment. In the early phases of the system life cycle, the different components of the MMS are outlined and introduced in the simulation environment as Athéna mock-up components. Each component can further individually evolve while keeping a global system simulation capability.

The simulation environment uses Athéna as a basis. A specific Athéna component has been developed to simulate the Internal Formation Data Link (IFDL)[3]. Figure 6.5 illustrates the way the different UCAV platforms are integrated in the simulation environment. Each platform uses the IFDL simulated component to communicate with other ones. Figure 6.6 shows how each platform is implemented. The different elements of a simulated UCAV are encapsulated in the platform component. An Aircraft Bus component has been developed as an integrated Athéna element. The different sub-systems of the UCAV are connected to this Aircraft Bus and use it to communicate with one another.

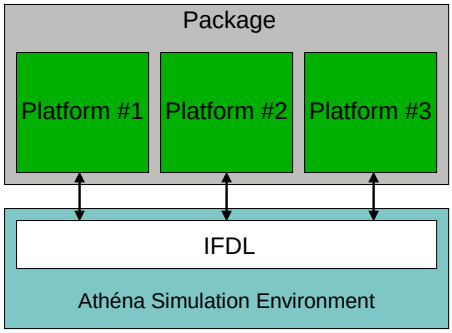

Figure 6.5: Platform integration in simulation environment

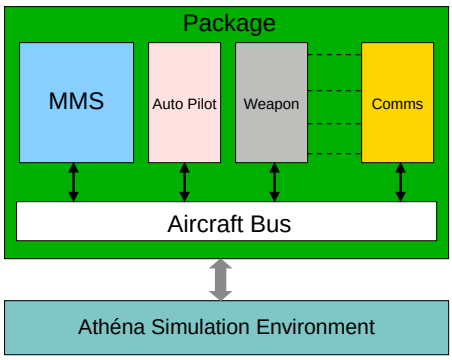

Figure 6.6: MMS intergration in simulation environment

In order to be able to see the results of the MMS activity, and particularily the results of the planning process, a small graphical interface has been developed. This interface is directly connected to the simulation environment. It polls different parameters and variables to draw the results of the planning process. Figure 6.7 shows a screen capture of the path planning visualization. The grey points are the waypoints of the mission, the blue points cercled by yellow circles are the threats (ground-to-air missiles and radars). The different color line are the paths used by the different package to reach the target area, at the bottom of the picture.

Following the Athéna principles, the focus is put upon the distribution of the decision through the package without changing anything to the simulation. We are able to transform all or a part of the Mission Management System of the number of UAV we choose and immediately check how these modifications influence the behaviour of the package in exactly the same environment.

[1]This work has been partially funded by the French Ministry of Defense through European and French projects.
[2]Timed portions of space in which the package is authorized to fly
[3]IFDL is a low-range data link used to communicate from one UCAV to another with low level of detection probability

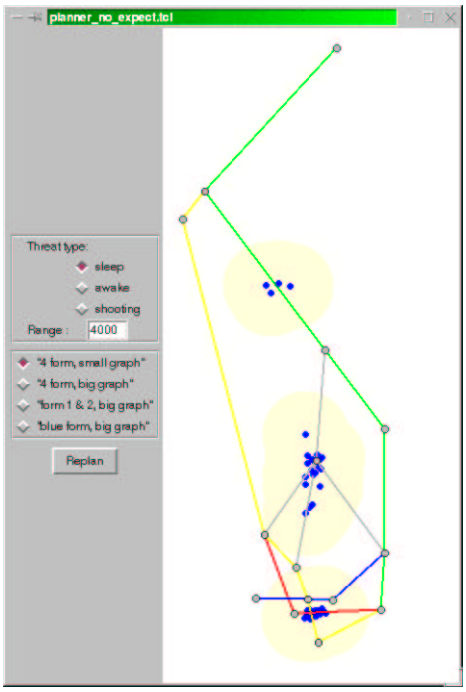

Figure 6.7: Package planning interface

Moreover, when the MMS will be a bit more stable, we will be able to incrementally introduce in the simulation other objects with a focus on piloted aircraft in order to assess the behaviour of the system with respect to the pilot.

6.2 Simulation of Embedded Real-Time Architectures

AADL (Architecture Analysis & Design Language) is an architecture description language initially designed for the avionics domain [13, 14]. Its purpose actually is more generic and it can be used for the description of any embedded real-time system. This language gives a means to describe system architectures with hardware and software aspects (processors, bus, memory, tasks, communication ports and connections, etc.). It is currently standardized by a committee under the authority of the Society of Automotive Engineers (SAE), aerospace division.

During a study with the European Space Agency, we have evaluated the first draft versions of this standard. To test the capability of this language to be supported by tools, we develope ADeS, a simulator of the behaviour of described architectures. Such a simulation can show the scheduling of the tasks, the state of the processors, the transmission of messages on the buses, etc. The idea of its realisation is to use the simulation kernel of Athéna with accurate high-level elements.

The component categories defined by AADL can not be directly mapped on the basic simulation elements provided by Athéna. Then, they have to be modeled by assembling several Athéna elements. This modeling is easy for some concrete components such as tasks or processors. A task is modeled by parameters representing its real-time properties and the progression of its execution, a processor uses an interaction to schedule the tasks and advance the execution of the most prioritary one.

Modeling data and event connections require more effort because this notion is distributed between a sender and receiver elements. For this category, and for all the other ones, the use of various specific parameters and interactions have enabled their representation in the simulation.

The control of the simulation kernel is performed by a specific user interface. Since the scenario description language designed by Athéna does not cover the AADL concepts, the user interface is also in charge of the explicit creation of the simulation elements on the server. During execution, each component is periodically activated to simulate its behaviour during a new time step. The user interface uses also CORBA to interact with the server, get the results and visualize them, as shown by figure 6.8.

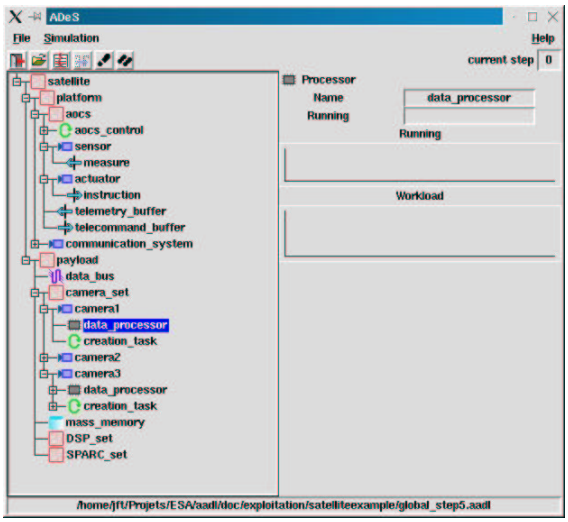

Figure 6.8: ADeS man-machine interface

This project has shown the real capability of the Athéna simulation kernel to support a simulation with a completely different purpose than its initial application domaine. Although the manipulated concepts are not the same as the initial Athéna ones, the adaptations has been easily done. They remains already feasible by any advanced user. That is why we never use any internal code, which would not be accessible for an external user.

7. Conclusion

Athéna is a flexible simulation and prototyping environment. Even if it was initially dedicated to aircraft autonomy, it has shown its capability to be used in various other domains (defence, space). Its high extensibility to support user components is its main strong point. This is illustrated by its use as a simulation kernel to build other simulators, such as ADeS.

In order to allow a smooth development curve a development club has been created (see [15]). The members of this club share Athéna and its generic components, and work together to improve it.

Within the context of this club, several improvements are foreseen. The main one is the addition of a native support for the autonomous agent notion. This new generic *agent* component will provide a simple way to represent agents, with knowledge management, decision autonomy, communication capabilities and other characteristics of the agent concept.

8. References

[1] http://escadre.cad.etca.fr/

[2] http://www.engenuitytech.com/products/STAGE/

[3] http://www.axlog.fr/prod/rtsim.html

[4] Adele HOWE, *Evaluating planning through simulation: an example using phoenix*, in Porceedings of the Workshop on the Foundations of Automatic Planning: The Classical Approach and Beyond, AAAI Technical Report SS-93-03, pp.53-57.

[5] T. WITTIG, N. R. JENNINGS and E. H. MAMDANI, *ARCHON – A framework for intelligent co-operation* IEE - BCS Journal of Intelligent Systems Engineering – Special Issue on Real-Time intelligent Systems in ESPRIT, 3(3):168-179.

[6] http://www.dassault-aviation.fr/athena/index.html

[7] Christophe GUETTIER, Bruno PATIN and Jean-François TILMAN, "Validation of autonomous concepts using the Athéna environment", ESA On-board Autonomy Workshop, Noordwijk, The Netherlands, October 2001.

[8] Claude BARROUIL, Bruno PATIN, Nicolas PREGO, "TANDEM: an agent-oriented approach for mixed system management in air operations", RTO meetings – Advanced mission management and system integration technologies for improved tactical operations, Florence, Italy, October 1999.

[9] http://www.cs.tu-berlin.de/~smile/ess97/

[10] http://www.corba.org/

[11] https://www.dmso.mil/public/transition/hla/

[12] Bruno PATIN, Stéphane NICOLAS and Jean-François TILMAN, "Athéna framework – Two examples of use in the aeronautic and space domains", ESA 7th International Workshop on Simulation for European Space Programs, Noordwijk, The Netherlands, November 2002.

[13] http://www.aadl.info/

[14] http://www.axlog.fr/R_d/aadl/aadl_en.html

[15] http://www.dassault-aviation.fr/athena/club.html

Author Biographies

JEAN-FRANÇOIS TILMAN is in charge of the system engineering activity at Axlog ingénierie. He has managed the development of the Athéna simulation kernel.

PHILIPPE MORIGNOT is chief scientific officer at Axlog ingénierie, Arcueil, France. He supervises and participates in the research & development projects.

BRUNO PATIN is the initiator of the Athéna development at Dassault Aviation and manages the activity of the Athéna user club.

JEAN-CLAIR PONCET is in charge of the autonomous system activity at Axlog ingénierie.

NELLY STRADY has participated to the development of Athéna and works for the autonomous aircraft studies at Axlog ingénierie.