Domain Specific Modeling for Operations Research Simulation in a Large Industrial Context

David Lugato
CEA/CESTA
33114 Le Barp, France
david.lugato@cea.fr

Marc Palyart
CEA/CESTA
33114 Le Barp, France
marc.palyart@cea.fr

Christophe Engelvin
CEA/CESTA
33114 Le Barp, France
christophe.engelvin@cea.fr

ABSTRACT
In order to conduct operations research studies on complex systems, CEA/CESTA[1] has been using and developing a new generation of simulator. The inherent complexity of such simulations ensues from the large spectrum of physical phenomena involved. As a consequence the description of a scenario and the analysis of parametric studies are time-consuming tasks. In this paper, we illustrate how the use of modeling techniques can help users construct and interpret simulation scenarios, especially users who are not computer scientists. To that end, we present a domain-specific modeling language for operations research. Moreover its associated tool, which is deployed in a large industrial context, is introduced along with model examples.

1. INTRODUCTION
The use of numerical simulations by manufacturers to improve the design of systems and to guarantee their requirements becoming more and more widespread. A set of numerical simulation software tools has been developed to validate individually each function of the systems designed at CEA/CESTA. But to find the best compromise between these various functions, we designed an operations research simulator that provides global parametric evaluations. These evaluations must meet the following objectives:

- Assess precisely the impact of a physical phenomenon on all or part of the system by using complex models designed by experts. These models may have different levels of detail and involve a large amount of data.
- Evaluate the impact that changing system parameters will have on the success of a given scenario.

The range of potential physics fields, detail levels, and behaviour possibilities is extremely broad, and many different skills are needed to fully model them. To build a scenario, users must establish the following variables: goals of each side, number and type of actors, strategy (motion path, behaviour, resource management, algorithms), hierarchy of communication systems.

[1]Commissariat à l'Énergie Atomique et aux Énergies Alternatives - French Atomic and Alternative Energy Commission

In 2005, CEA/CESTA decided to design a new operations research simulator. In a nutshell, the requirements for this new simulator were:

- *Unique Platform*: able to handle the various stages of the simulation process (actors modeling, scenarios definition, simulations execution, results analysis) within the same software.
- *Abstraction*: users are physicists not computer scientists, modeling actors and creating scenarios must be as simple as possible and easily understandable.
- *Capitalization*: capture and share the knowledge and skills of several physicists.
- *Maintenability*: insure a life expectancy superior to twenty years for models.

To fulfill these requirements we decided to rely on model-driven engineering techniques. The choice of MDE seemed natural as many of these requirements are now beginning to be recognized as MDE benefits in the literature [10]. In addition this development was part of a broader program initiative launched at CEA/CESTA to introduce MDE to different domains such as high-performance computing [9, 14] and graphical user interface for dataset edition[12].

This paper is organised as follows: first we describe the overall architecture and process of the simulator (Section 2); secondly we present and explain our domain-specific modeling language for operations research (Section 3); finally in Section 4 we discuss similar projects and identify potential avenues for future work.

2. SIMULATOR OVERVIEW
The simulator results from the combination of a platform with several models. As a matter of fact the platform act as a host which manages the interaction of different actors based on an initial scenario. The platform is able to generate code from models and execute them on-the-fly. In this section we first describe the general architecture we chose to adopt for the development of this simulator. Then we present the different phases that make up the simulation process. Finally we outline the execution model.

2.1 General Architecture

The execution engine of the platform manages the simulated time. Time scheduling is subject to the production of events. We voluntarily chose a simple design based on a *discrete events communication* paradigm [2] for the simulation execution. This choice was made for three key reasons: it is formally and simply defined, users rapidly understand it and above all the previous simulator was based on this paradigm.

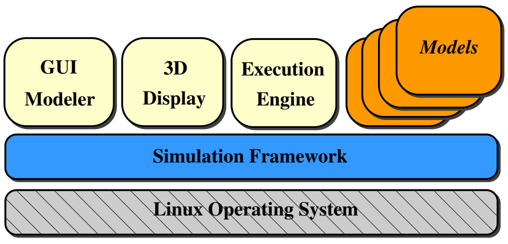


Figure 1: Simulator General Architecture

In discrete systems, state variables change only at discrete points of time, even the global time variable is discrete. During the simulation, the execution engine takes into account messages (or events) sent by actors, stores them and distributes them according to their chronology. These messages must therefore be dated prior to the current simulation time.

The execution engine also features all of the services users expect from such simulation platforms: configuration management, debugging, scripting, GUI customisation, automatic documentation generation. The general architecture of this platform is shown in Figure 1. The design approach illustrated here is based on the notion of re-usable software components.

We will not present in this paper all the functionalities and implementation details of the simulation platform but should indicate that the platform is mainly implemented in C++. Graphical user interfaces rely on the wxWidgets toolkit.

2.2 Simulation Process

The overall simulation process is introduced in Figure 3, using the SPEM notation [13]. The global process is made of two sub-processes, the one on the left concerns the definition of models by domain experts and the one on the right concerns the instantiation and execution of models by operations researchers.

Even though we face two different profiles, a same person may play these two roles. The choice of a clear separation between the activities of definition and use of models was made to enhance knowledge capitalization and reusability. Furthermore, models are stored either in personal or shared databases in order to stimulate collaboration between experts from different fields of physics, for example sensors experts will not model flight mechanics themselves but can use stored flight algorithms in their own scenarios. Each step of this process can be performed within the simulation platform from four different perspectives:

- *Modeling Perspective*: specification of actors by using structural and behavioral models.

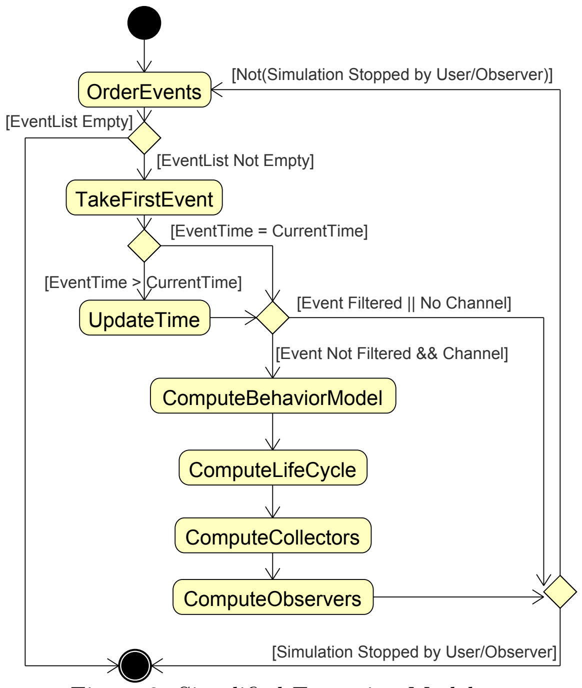

OrderEvents
[Not(Simulation Stopped by User/Observer)]
[EventList Empty]
[EventList Not Empty]
TakeFirstEvent
[EventTime = CurrentTime]
[EventTime > CurrentTime]
UpdateTime
[Event Filtered || No Channel]
[Event Not Filtered && Channel]
ComputeBehaviorModel
ComputeLifeCycle
ComputeCollectors
ComputeObservers
[Simulation Stopped by User/Observer]

Figure 2: Simplified Execution Model

- *Scenario Perspective*: definition of scenario composed of instances of valid actors. These actor models can either come from a personal or shared model database.
- *Execution Perspective*: execution of scenario with visualization in 3D environment. User can control the execution engine through a media player-like interface (play, pause, stop, step by event, step by time).
- *Analysis Perspective*: analysis of results from simple or parametric study. User can define data mining functions, execute them and store results in several file formats (csv, netCDF, html).

In addition to these perspectives, the simulation framework implements user profiles in order to adapt the GUI and the framework services more closely to users' needs.

2.3 Execution Model

The general execution model of the simulation platform is described in Figure 2. It is important to notice that there is no parallelism between the different executions of actors' behavior such as that encountered in multi-agent system simulation. Within a scenario events are computed one by one, sequentially. Even though this choice may appear as strongly restrictive in terms of performance, two main reasons motivated the choice of this execution model. On the one hand, the sequentiality of events implies easier modeling and debugging of behavioral models. On the other hand, the simplicity of behavioral models executed within the platform when compared to a specific real scale actor simulation (e.g. a simulation that assess how a radar performs) means that this system requires only relatively modest computational resources. However parallelism can be exploited at a higher level with parametric studies which sometimes involve thousands of scenarios.

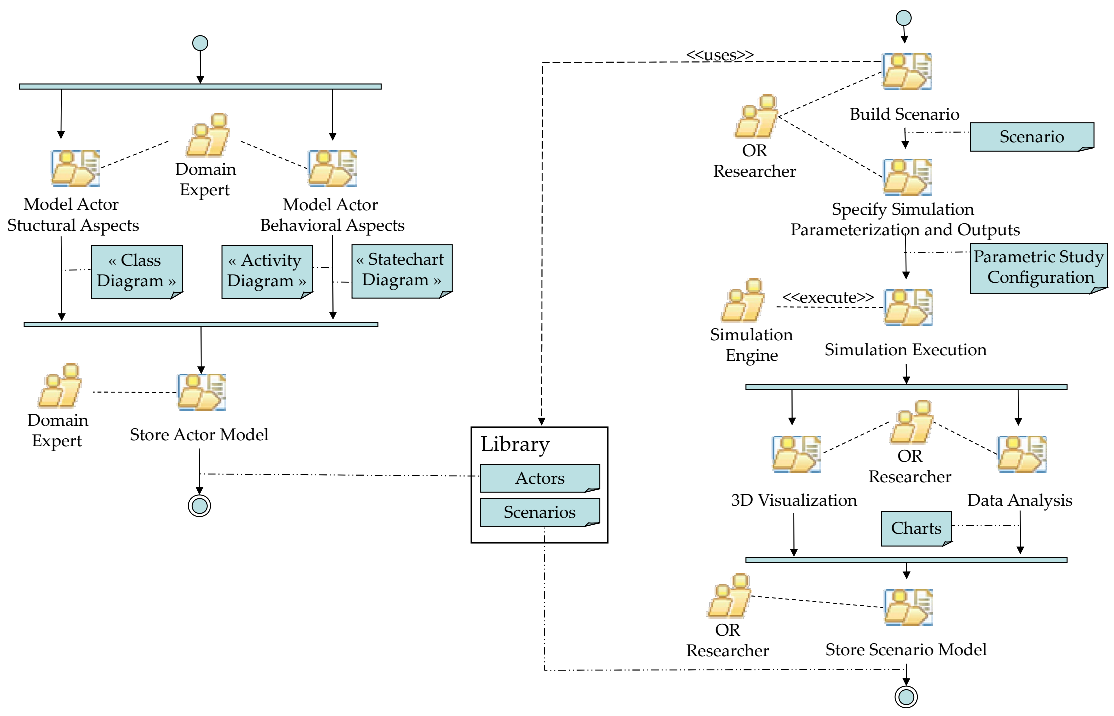


Figure 3: Simulation process

Another particularity of the execution model is that there is no human intervention in the execution process. The user first defines a scenario composed of actors, then starts the execution. The user can only stop the execution or put the scenario on pause to examine the current status of individual actors and variables. There is also no interaction with real actors (e.g. a real sensor/device or parts of it) during the execution, we do not manage the concept of federation of HLA (High Level Architecture) defined in the IEEE 1516 norm [15].

3. MODELS PRESENTATION

The design of a domain-specific modeling language (DSML) is a complex and tedious task which requires the formalization of several aspects of the language. For instance, the definition of the concrete syntax which acts as a go-between concepts and users is essential, unfortunately this step is frequently neglected [11]. Similarly the definition of clear semantics [6] plays an important role but requires an important investment. In order to ease this process and to reduce design errors, we decided to start from well defined languages. In this section we detail the different modeling phases introduced in Section 2 by presenting each associated metamodel.

3.1 Structural Models

The first part of the DSML is dedicated to the definition of the structural decomposition of each actor. This definition relies on a formalism based on UML class diagrams with numerous syntactic and semantic restrictions compared to the UML standard. We have opted for a classic use of this formalism to represent the structure of the actors' static

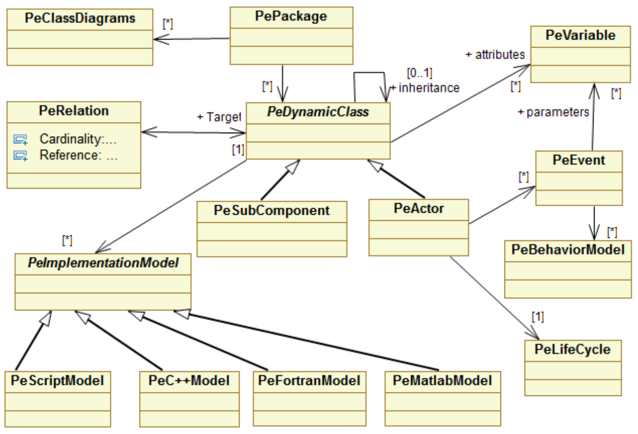


Figure 4: Simplified class diagram metamodel

data and their sub-components. The available constructions have, however, been considerably restricted to simplify the work of users, who are not computer scientists.

The Figure 4 shows the simplified metamodel dedicated to structural modeling. Main constructions are Classes (called *PeDynamicClass*), Packages (called *PePackages*), simple inheritance and associations (called *PeRelation*). The only difference between *PeActor* and *PeSubComponent* comes from the fact that *PeActors* have one *PeLifeCycle* and can receive *PeEvents*. The Figure 5 presents an instance of this metamodel with a classic decomposition of a *Carrier* system and its sub-systems.

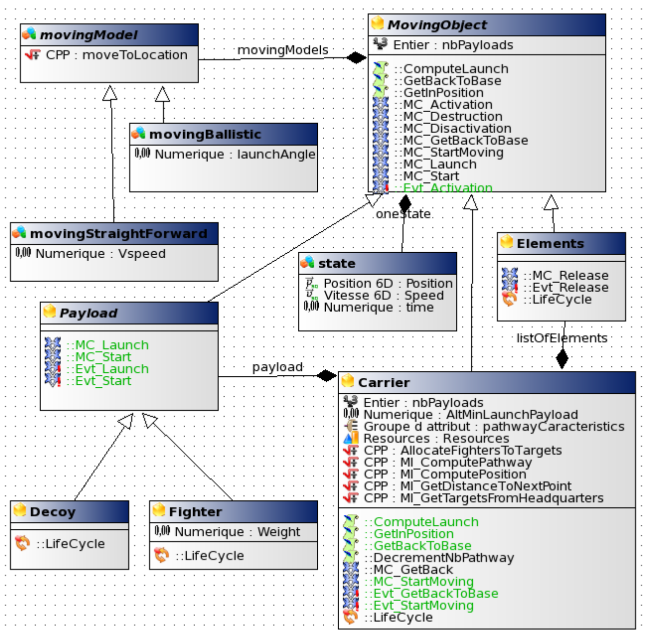


Figure 5: *Carrier* class diagram model

As mentioned by Moody in [11], we observed that UML graphical notations lack of cognitive effectiveness. Hence, we chose to add one more visual dimension with a semantic use of color in our diagrams. For example in our class diagram formalism (see Figure 5), sub-classes do not show their inherited attributes. We also use the black color to identify new elements (event, attribute, behavior model, lifecycle) in an actor. The green color represents the override of these elements (changing default behavior, changing default value, redefining behavior model or life cycle).

3.2 Behavioral Models

The first step in defining an actor's behavior is to model its general state. We chose to adopt a simplified version of Harel statecharts [5]. We use the concept of hierarchically-nested states but we have removed the concept of orthogonal regions. The statecharts diagram are called *lifecycle diagrams* in the simulator. The Figure 6 shows their simplified metamodel. The Figure 7 presents an instance of this metamodel with the description of the *Carrier* actor life cycle (see Figure 5).

Guard conditions are boolean expressions which are evaluated dynamically according to actor variables, events and time conditions. A tailored script language is used for the definition of entry and exit actions. Notice that there are no actions on transitions and no possibility to send events on transitions.

Lifecycle diagrams enable to avoid useless in-then-else instructions in the implementation models. Event filters ease the modeling of sender actors because they do not have to worry if receivers are dynamically able to execute the event.

The execution model of the life cycle diagram is based on a *Run To Completion* (RTC) algorithm. At each stage of the simulation, all executable transitions are carried out until the life cycle reaches a stable state with no executable transition. We have implemented static validation rules that check that a life cycle is syntactically correct and does not have too many states or transitions. However no complex dynamic validation such as detection of concurrent transitions or possible infinite loops is implemented. We believe that a life cycle diagram should be simple in order to be rapidly understood by all users and to avoid the need for such complex validation.

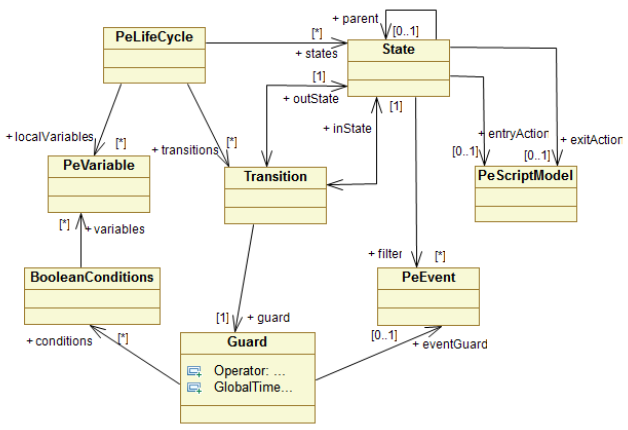


Figure 6: Simplified lifecycle metamodel

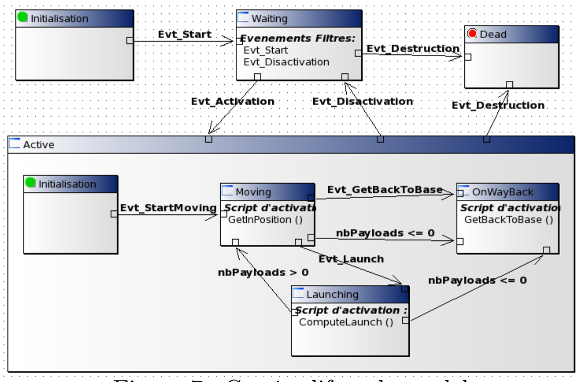
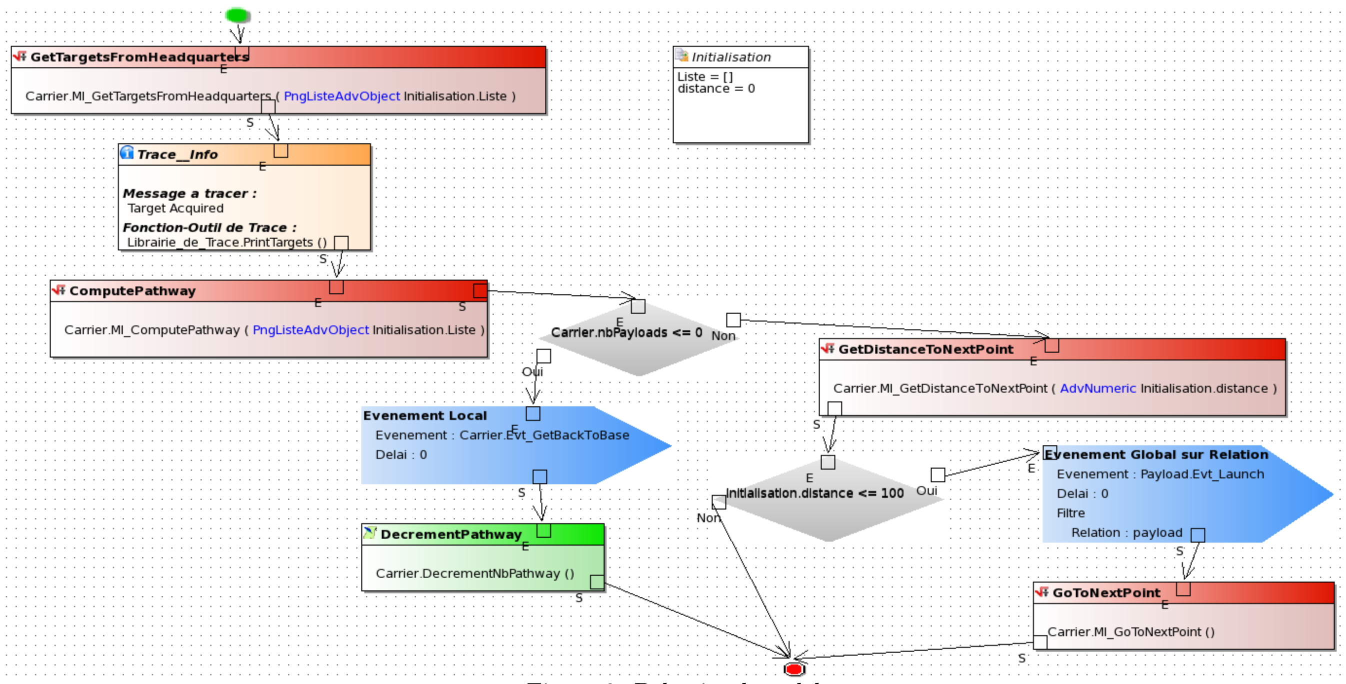


Figure 7: *Carrier* lifecycle model

The second step for defining an actor's behavior is to specify events creation with one or several *behavioral models*. We chose to adopt a simplified version of the *UML activity diagram* to represent these behavioral models. The level of modeling is extremely important. It must be as detailed as possible while remaining sufficiently abstract to be rapidly understandable by users from different domains of expertise. Consequently, only a few constructions are available in the activity diagrams : send events, call implementation models, add debugging traces, define branch conditions. Activity diagrams offer hierarchical levels in the same way as the life cycle diagrams.

The Figure 8 shows the simplified metamodel of these activity diagrams. The Figure 9 presents an instance of this metamodel that describes the StartMoving model of the *Carrier* actor introduced in Figure 5).

A *send event* statement(blue arrows in Figure 9) can send an event either to one actor instance or to all of them. Two kind


GetTargetsFromHeadquarters
Carrier.MI_GetTargetsFromHeadquarters (PngListeAdvObject Initialisation.Liste)
Initialisation
Liste = []
distance = 0
Trace__Info
Message a tracer :
Target Acquired
Fonction-Outil de Trace :
Librairie_de_Trace.PrintTargets ()
ComputePathway
Carrier.MI_ComputePathway (PngListeAdvObject Initialisation.Liste)
Carrier.nbPayloads <= 0
Non
Oui
GetDistanceToNextPoint
Carrier.MI_GetDistanceToNextPoint (AdvNumeric Initialisation.distance)
Evenement Local
Evenement : Carrier.Evt_GetBackToBase
Delai : 0
Initialisation.distance <= 100
Evenement Global sur Relation
Evenement : Payload.Evt_Launch
Delai : 0
Filtre
Relation : payload
DecrementPathway
Carrier.DecrementNbPathway ()
GoToNextPoint
Carrier.MI_GoToNextPoint ()
Figure 9: Behavioral model

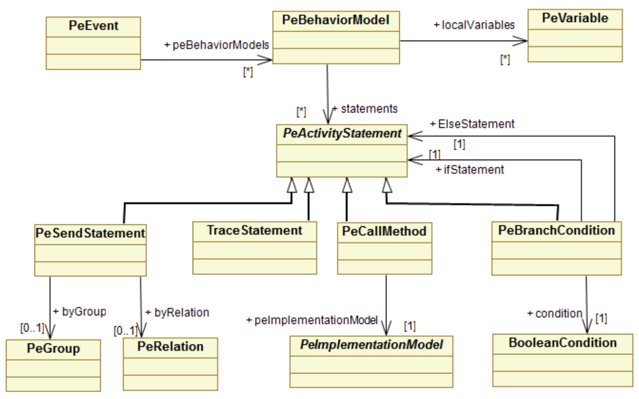


Figure 8: Simplified behavioral metamodel

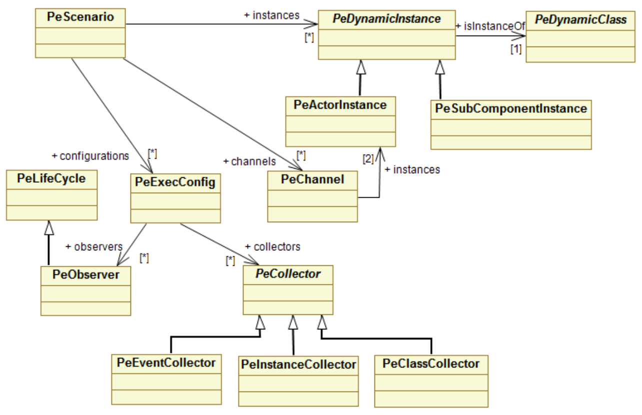


Figure 10: Simplified Scenario Metamodel

of filters can be added to this statement: relation filter (the event will only be sent to all actor instances that are present in the relation) or group filter (the event will only be sent to members of a group defined in the scenario). Inheritance is also useful to send events to a group of similar actors.

The implementation models can be developed with four different languages: C++, *Fortran*, *Matlab* (shown as red boxes in Figure 9) or a *Python*-like script language (shown as green boxes in Figure 9). The mapping between the actor variables and the parameters is automatically generated by the simulation framework.

The debugging traces, shown as orange boxes in the Figure 9, are functionally tagged (defense, threat) and by levels (warning, error, trace, validation). As a consequence, users can decide which kind of traces they needs before executing a scenario.

3.3 Scenario Models

Once actors models are complete (structural, lifecycle and behavioral), users can switch to the perspective of the simulator dedicated to scenarios building. A scenario is a set of actor instances, execution configurations and communication channels.

An execution configuration defines the set of variables and events that need to be collected. Each collector writes its output in a dedicated file. Several formats are available : *xls*, *netCDF* or *HTML*. Users can also define observers with lifecycle diagrams. These observers enable the definition of complex suites of actions (events reception or boolean conditions on the instance variables). It then is possible to use them as a symbolic breakpoint in a scenario.

Communication channels allow two actor instances to exchange events. They act as filters as they condition the exchange of message between two actor instances.

The Figure 10 shows an extract of the metamodel for scenarios building. The user can create new instances of actors or duplicate them from his scenario database. One scenario can have several *PeExecConfig* in order to assigned collectors to a specific task (debugging, tracing, analysing).

The execution engine is coupled with a 3D interface that offers the following services: help for building scenarios (such as locations of instances or communication channels), monitoring facilities for the execution of the simulation and finally tools for results analysis. Simulation data can be included in the display (event, position, speed, gradient, bearing, sensor field). The objects shapes are representative of their real geometry. The display is highly adjustable: viewpoint orientation, zoom on specific areas, object trajectories, cameras positioning, scenario staging. All this graphical information is represented with the help of a dedicated metamodel that we will not present in this paper.

4. DISCUSSION AND PERSPECTIVES

Research into modeling and simulation applied to the design of complex systems frequently relies on the DEVS formalism [16]. In [8] Kim et al. present such work, but unfortunately they use models mainly as descriptive entities. Regarding other application domains, several experience feedbacks (telecommunications [1], avionics systems [3], software migration [4]) on the use of model-based approaches in industrial projects record improvements in productivity and software quality. Closer in spirit to our work, Kienzle et al. in [7] described that the use of models to specify the behavior of non-player characters(NPC) in computer games such as *Tank Wars* can have many advantages : appropriate level of abstraction, correct formalism and visual notation, enhanced modularization, evolution and reuse.

In this paper we have presented a domain specific modeling language and its associated tool that are used in an industrial context. At the moment we are in the process of assessing the impact of choosing a DSM approach in term of productivity. We plan to compare the model-based simulator with the previous simulator developed with a more ”classic” approach. This evaluation will be based on several metrics regarding developments costs, maintenability and accessibility. However we can already state that the choice of MDE for our simulator development has been a winning strategy. For example, three different roles were necessary in the old simulator: developers, end users and expert users that were the only ones able to make scenarios due to the complexity of the tasks involved. With our new simulator and thanks to the accessibility of the domain specific language, the team is only composed of developers and end users as all scientists can now build scenarios by themselves. Furthermore users can now focus on their domain of expertise and build more complex models. It is now possible to define and analyse large parametric case studies (up to 10 000 cases).

This positive experience reinforces our confidence in domain specific modeling techniques. For this reason we are also in the process of defining a DSML to simplify the results analysis of parametric studies.

5. REFERENCES

[1] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context: Motorola case study. In *Model Driven Engineering Languages and Systems*. 2005.

[2] J. Banks, J. Carson, B. L. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 4 edition, Dec. 2004.

[3] T. Bloomfield. MDA, meta-modelling and model transformation: Introducing new technology into the defence industry. In *Model Driven Architecture Foundations and Applications*. 2005.

[4] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-driven engineering for software migration in a large industrial context. In *Model Driven Engineering Languages and Systems*. 2007.

[5] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of ”semantics”? *Computer*, oct. 2004.

[7] J. Kienzle, A. Denault, and H. Vangheluwe. Model-based design of computer-controlled game character behavior. In *Model Driven Engineering Languages and Systems*. 2007.

[8] J. Kim, C. Choi, and T. Kim. Battle experiments of naval air defense with discrete event system-based mission-level modeling and simulations. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 8(3):173, 2011.

[9] D. Lugato. Model-driven engineering for high-performance computing applications. In *International Conference on Modelling and Simulation*, 2008.

[10] P. Mohagheghi and V. Dehlen. Where is the proof? - a review of experiences from applying MDE in industry. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, 2008.

[11] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35:756–779, November 2009.

[12] D. Nassiet, Y. Livet, M. Palyart, and D. Lugato. Paprika: Rapid UI development of scientific dataset editors for high performance computing. In *SDL 2011: Integrating System and Software Modeling*, 2011.

[13] Object Management Group. Software Process Engineering Meta-Model, version 2.0. Technical report, 2008.

[14] M. Palyart, D. Lugato, I. Ober, and J. Bruel. Improving scalability and maintenance of software for high-performance scientific computing by combining MDE and frameworks. In *Model Driven Engineering Languages and Systems*, 2011.

[15] (SISC), Simulation Interoperability Standards Committee. IEEE Standard for Modeling and Simulation High Level Architecture (HLA) - Framework and Rules. Technical report, 2000.

[16] B. Zeigler, H. Praehofer, and T. Kim. *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. 2000.