Towards a PIM for virtual prototyping

Adel SGHAIER, Thierry SORIANO, Skander TURKI
Mechatronics department
LIISIM SUPMECA
Maison des technologies 83000 Toulon
FRANCE

Abstract: We work on the way of modelling industrial machines for virtual prototyping. We use the OpenMASK environment as a tool for building the prototype. The whole virtual model of a machine is composed by the association of modular generic objects. The association of modular parts is considered from the geometrical and the behavioural points of view. In this article we present a solution for virtual prototyping based on the UML language and the MDA architecture. We present first the PSM (Platform Specific Model) build with the Objecteering UML tool and then we describe the bases of the future PIM (Platform Independent Model).

Key-words: UML 2.0, MDA, virtual prototyping, hybrid automaton, Openmask, Grafcet

1 Introduction
Companies want today to master their products lifecycles to maximize their competitiveness in the market. Reducing the design and manufacturing delays is still to be improved as companies must enhance their reactivity especially in the decision making step. Virtual prototyping is one of the tools that help establishing a strategy that aims to detect the weaknesses of a product in an early stage of its lifecycle.

We work on the way of modelling industrial machines for virtual prototyping. We use the OpenMASK environment as a tool for building the prototype. The whole virtual model of a machine is composed by the association of modular generic objects. The association of modular parts is considered from the geometrical and the behavioural points of view. In this article we present a solution for virtual prototyping based on the UML language and the MDA architecture. We present first the PSM (Platform Specific Model) build with the Objecteering UML tool and then we describe the bases of the future PIM (Platform Independent Model).

2 MDA
The UML language is a toolbox that was built to help establishing a blueprint of an application, to help estimating and planning the development process and to allow the communication between teams. UML was then extended with different profiles. A profile is a group of coherent stereotypes, tagged values and constraints, such as the UML real-time profile that contains specialised elements that help represent a system with real time requirements. The large number of UML profiles made of UML a very large toolbox. The communication purpose of UML become hard to obtain. But the OMG is working on the UML 2.0 and we expect the different profiles to be standardised. This will allow us to use the UML language as a stable low-level communication layer. The UML [2] [4] language gave a solution to facilitate the analysis and the design of applications. But, as it is just a tool, the MDA solution is a very appropriate and efficient layer that can help and guide using UML. It's another abstraction to software development and may help having systematic processes to develop specific software.

The OMG proposed an approach to the development of software, the MDA (Model Driven Architecture). This approach consists of using models and model transformations. It is based on different standards and concepts of the OMG like UML, MOF (Meta-Object Facility) [2], CWM (Common Warehouse Meta-model). It consists of the separation of the PIM (Platform Independent Model) from the PSM (Platform Specific Model), which allows the application designer to have a better abstraction and an easier reusability of his components.(Figure 1) [3] Applying MDA is first done around a PIM that represents an accumulated know-how on the specific field. Second, models built on this PIM will be automatically transformed to PSMs that represent know-how accumulated on specific platforms. Finally documents or code are generated from these models automatically.(Figure 2) The MDA architecture is supposed to be applied inside a company to organize a global development process that includes all the
development processes of projects related to a specific field. But we cannot just dump and MDA architecture application on a company an expect it to run perfectly. The integration of MDA needs to be planned, it represents a complete project on its own. The PIM must first be built taken into account a minimum number of functionalities. The PSM is built upon this first PIM. This architecture is tested inside the development process of the current project. The feedback of this first experience will help improving the first PIM by integrating new functionalities, by adding consistency check tests to the PIM construction step, by improving the mapping between the PIM and the PSM and by adding automation to the code generation step. These models are built and improved gradually inside a process meant to integrate this architecture inside a company. Separating the PIM from the PSM is an essential element in the MDA philosophy. In the case of virtual reality environments, for example, the existent platforms are coexisting in the platforms market; we cannot predict the future of these platforms. It is not acceptable for our projects to rely on a specific platform. We must be able to switch from a platform to the other whenever it is useful to do so, with the minimum of effort.

![Fig. 1: MDA global view](image)

3 Openmask

OpenMASK is an open source platform developed by IRISA (Rennes France). It has a modular architecture; it means a virtual reality application created with this tool is composed of modules connected to a data bus. A Virtual environment described with OpenMASK is organised in a simulation tree. Any description begins with a root object which is the execution controller. Then the simulation tree is composed of modular elements (simulation modules) [5]. We describe the parts of the VR application by modular objects with some specialised methods. The most important methods to specialise are the method of the initialisation (int()), the method of simulation step, the behaviour of modules is updated by the controller (the code in the compute method is executed) with respect to the frequency of the module defined in its description. The communication in OpenMASK is realized over the data bus by means of the input/output interface of the modules. Each module can have inputs/outputs, control parameters connected over the data bus,
behaviour computing (\texttt{compute()}) and the method of event handling (\texttt{ProcessEvent()}). The behaviour is programmed in C++ into the compute method. The way to program the behaviour of an object is completely left to the user’s choice. Two modules can communicate directly with connecting the output of the producer module to the input of the consumer module. This way of communication is the data flow communication approach. Another way to communicate in OpenMASK is by sending events: a producer module can send an event to the consumer module and run some methods associated to this event. Finally, a module can send a signal through the data bus. A signal is exactly like an event except the fact that it does not have an address; all modules registered to receive this signal receive it and can react to it. The OpenMASK environment is supplied with an interface to the SGI Performer rendering engine. It enables animation of visual objects represented with 3D models like VRML models. We can distinguish two different aspects of object description in the virtual scene: The geometry and the behaviour. Every Object in OpenMASK is represented by a C++ class which carries the behaviour and the communication linked to a geometric file [5]. The modular architecture of OpenMASK can be very useful to create a virtual machine by assembling modular objects. So we may use this concept of modules to represent the model of a physical part of the machine to be modelled.

4 The PSM produced

In our case each PSM will be related to a particular virtual reality environment and the PIM will be a global model, from which we could express the subtleties of all the platforms: From the virtual reality environments, we can extract models or PSMs using a reverse engineering (rising arrows in Fig 2). Each of these models must represent the entire simulation environment associated to it. A second step will help obtaining a PIM that generalises all the PSMs. This PIM will be used by developers to build their simulation applications.

Descendant arrows represent the development process of a particular application in a virtual reality environment. This direction requires the definition of the transformation mechanisms that will give the PSM based model out of the PIM based model. It also requires the code generation rules. We present now the construction of a PSM for OpenMASK, and the code generation related to it.

4.1 The Objecteering/Profile Builder solution

The Objecteering Profile Builder [7] is a tool that can be used to build profiles. A profile is an extension of the UML language using some specific mechanisms which are the stereotypes, tagged values and constraints. Stereotypes are meant to extend the semantics of a meta-class, per example, a class that is an abstract class will be stereotyped abstract; tagged values are used to add a parameter for a meta-class; constraints are used to limit the scope of a meta-class [6][7]. This profile is used to create an analysis and design integrated environment with a personalised meta-model. A meta-model is the description of the language itself. This design integrated environment is used then to generate code in a personalised way. With the Objecteering profile builder, we can define the meta-model of the profile we are creating and immediately use the extensions defined inside an application. The use of this tool resolves the problem of the improvement of the models. In fact, it is easy to introduce new know-how in our models and test them in an application. This solution supports easily an iterative and incremental process of the building of the models related to the domain.

It is also true to say that this tool presents some weaknesses like its use of a proprietary language (the J language) which requires a hard learning. In addition it doesn’t have a debugger which complicates a lot the coding part. Yet, compared to other solutions, this is the more suitable to implement an MDA based approach [8].
OpenMASK profile is composed of the design (figure 3), the analysis and the code generation sub-profiles. The analysis sub-profile is not treated in this work. The elements used in the design sub profile of OpenMASK are, essentially, an state diagram.

4.2 The OpenMASK Design sub-profile.
The extended class diagram will be used first to add simulation objects, their attributes and methods, then to describe the attributes stereotypes and tagged values and constraints. The class diagram will also be used to add predefined modules of the OpenMASK library that add new functionalities in the application, and that just need to be parameterised such as the PsKeyboardNavigator object that introduces the keyboard navigation inside the virtual scene. The extended state diagram will be used to define the dynamic behaviour of the simulation objects. We defined two tagged-values related to the state meta class, the first is the equation evolution tagged-value. It is used to define the dynamic behaviour equation of objects using the values of attributes in the previous simulation step. And the second is the differential equation tagged-value that is used to declare the differential equation of the object. A constraint is also defined here, it prevents a simulated object to have more than one tagged-value of these.

3 Towars a PIM.
Our general objective is to develop virtual prototypes of industrial assembly machines. For that, we want to create modular virtual components which will be assembled to form a virtual machine. So we use the modular concept of OpenMASK to implement the different parts of the machine. The main question here is how to model a modular physical part of a machine (like a jack or a motor…). The virtual components will be modelled with two parts. First, the geometric part which will be described in a VRML file will be handled by the render engine (Open Performer). Then, the behaviour of the object will be modelled separately from the object (like the geometry). The benefit of this way of modelling the behaviour is that we can model the behaviour of an object independently from the object itself, so we can change the behaviour of an object without rewriting the whole class of the object. A second identified requirement is to model the behaviour using a high level of abstraction formalism. As it can be observed in the description of OpenMASK

![Fig. 3: The OpenMask design sub-profile:](image-url)
extension, the places of the Petri net has continuous token [9]. Finally, we can use the hybrid automata formalism to describe the hybrid behaviour. The hybrid automata are an extension of discrete automata. It is the association of classical automata (discrete) and set of differential equations (continuous). This method enables us to easily model a hybrid system with general continuous behaviours thanks to the differential equations. Our choice has been the hybrid automata formalism because of its generality [10]. So we will model the behaviour by an external hybrid automaton.

4 Assembly of virtual parts
The virtual parts obtained using the method described previously must be assembled with the others to form the virtual machine. As we have modelled a virtual component above in two part, a geometric part and a behavioural one, we have to consider the assembly of the virtual components from the two points of view.

4.1 From the geometric point of view
The geometric representation of the different parts composing the virtual machine must be geometrically assembled. We assemble these geometric parts statically in the Performer scene by passing their position to the simulation file by the mean of an argument script file. The use of this file has the advantage of avoiding the compilation of the source each time we change the objects static coordinates. The dynamical evolution of the geometric parts is unidirectional for the moment. This means that it is controlled in one direction by the behaviour but there is no return of data from the geometry model to the behavioural model. Nevertheless, we think they will be two ways to do it. We distinguish two types of interactions from the geometry to the behaviour. First, the collisions between virtual parts and second the cinematic relationship between parts. For the task of handling the collisions between parts, we have begun a work on integrating the Vcollide method in OpenMASK. The Vcollide is a method for collision detection between VRML files. In another side, we are waiting for the new version of OpenMASK which will include collision handling objects. For the virtual parts cinematic relationship implementation, there is a work on creating OpenMASK objects which will be responsible of this task using the Open Dynamics Engine (ODE) libraries. Using this method, we can model cinematic relationship by the means of joints and constraints.

4.2. From the behaviour point of view
The behaviour of the virtual machine is divided in two parts. The control part and the operative part. The operative part is the association of the behaviour of all the virtual parts which are represented by hybrid automata. The control task is performed in the real machine by the industrial automata using the GRAFCET (or SFC Sequential Function Chart) formalism. This is why we will use in the virtual machine the GRAFCET formalism to represent the behavioural part which will command all the automata of the virtual parts.

The conception of the behaviour of the virtual part is as follows. First, we define the behaviour of each part of the virtual machine by a hybrid automaton; this hybrid automaton will be used to generate the corresponding OpenMASK behavioural objects. We can reuse the description of a type of generic virtual part in the virtual machine assembly. So if for example, we use 3 jacks in the machine, we don’t have to describe the jack three times but we can reuse it. The second step in the conception of the virtual machine is defining the GRAFCET of control of the machine. Since we model a real machine we can directly use the GRAFCET of the real machine if the virtual machine is faithful enough. To make the GRAFCET able to command hybrid automata we must transform it into a hybrid automaton. The method that we use to translate the GRAFCET to hybrid automata consists in using the situation graph to determine the states and the GRAFCET for the conditions (invariant and guards) of the automaton. We use then composition of hybrid automata to implement the control. This control is realised through the composition of hybrid automata which consists of synchronizing two or more transitions in two or more hybrid automata. It means that if a transition in the control automaton holds the same label of one or more transitions in the hybrid automata of the parts, the control automaton transition entails the firing of the controlled automata transition. We are currently implementing this method.

3 Conclusion
The UML [2] language is a good solution that helps with the development of software. But, as it is no more than a language and not a method, we need in
addition to it another layer like the MDA architecture to establish a systematic process that minimises the time-to-market of software. This systematisation is vital in virtual prototyping with a target tool as Openmask. We have proposed a complete experimentation of a PSM building and we have presented the high level abstractions, based on geometry and behaviour, which are the bases of our future PIM.

References