The SimTG Simulation Modeling Framework

Olivier Zanon[(1)], Ambros Morscher[(2)]

[(1)]ASTRIUM Satellites
31 rue des Cosmonautes
Z.I. du Palays
F-31402 Toulouse Cedex 4
Email: olivier.zanon@astrium.eads.net

[(2)] ASTRIUM Satellites
Claude-Dornier-Str.
D-88090 Immenstaad BW
Email: ambros.morscher@astrium.eads.net

INTRODUCTION

During the last years, Astrium Satellites achieved a major development in the frame of simulation. Teams from Germany and France gave birth to SimTG, the new generation of simulation infrastructure (SimTG is now operationally in use in all spacecraft programs).
However, one last stage was to be achieved: the development of a harmonized modelling workbench. This development started in 2009. It was a shared effort between SimTG teammates across Astrium countries. The **Simulation Modelling Framework** or **SimMF** is now to be operationally released.
The purpose of SimMF is to offer a complete model development workbench dedicated to simulation. It promotes process unification through teams and countries, and is meant to be used by every simulation model engineer within Astrium Satellites.
It leverages the best of Model-driven technologies to support the development of spacecraft system simulators.
In this paper, we will introduce the technologies used by SimMF and the innovative features that SimMF brings.

CONTEXT

SimMF currently supports following steps in simulation development:

- Simulation design :
  - hierarchical decomposition of models into sub-models
  - creation of member variables
  - creation of model operations
  - creation of connections between models
- Simulation production :
  - c++ based realization of simulation models by using code-generation
  - automatic creation of model libraries
- Simulation documentation : establishment of documentation material reflecting model design

As development goes on SimMF is planned to provide support for model specification and model verification as well.
Following figure shows the SimMF context according to its provided functionality.

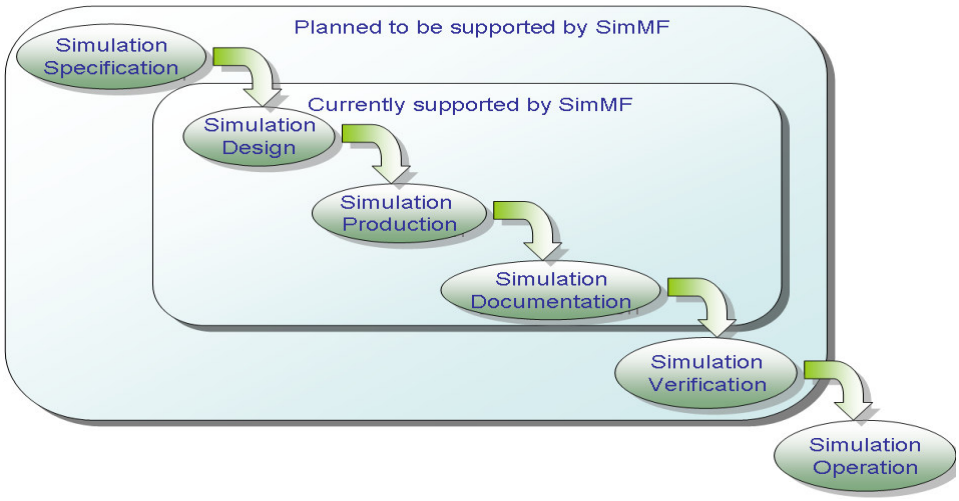

Fig. 1. SimMF context

METAMODEL BASED SOFTWARE

SimMF heavily relies on the Model Driven Architecture. The SimMF metamodel is expressed both in OMG MOF[1] (by means of UML class diagram), and Ecore from the Eclipse EMF[3] project.

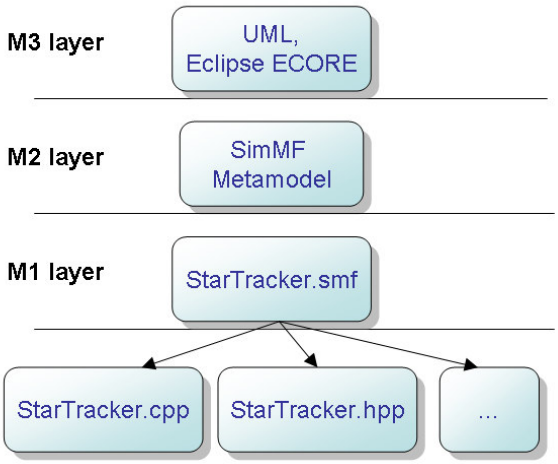

Fig. 2. Ecore and the SimMF metamodel in the Meta-Object Facility architecture

To benefit from all MDA-oriented projects of Eclipse, a metamodel has to be expressed in Ecore. Transformation from a UML class diagram to Ecore is straightforward, since Ecore is no more than an implementation of the OMG EMOF (Essential MOF).

The SimMF metamodel describes all kinds of data manipulated and optionally stored by the SimMF application and the relations between these data.

One of the major instances of the SimMF metamodel is the model file (.smf file extension), as seen in figure 1. It describes a single simulation model. The following figure shows the core part of the SimMF metamodel. Instances of those classes are stored in model files.

The following figure shows the core of the SimMF metamodel. It works as following.

- The *Model Definition* is the top node that describes a model. A *Model Definition* can contain many *Property Definitions*, many *Model Usages*, and many *Connections*. *Property Definitions* are also called *ports*, if they are an *input* or an *output*.
- *Ports* define the interface of a simulation model, its input and output data that are meant to be connected or simply observed.
- *Model Usages* are instantiations of *Model Definitions*. They offer the same *ports* (*Property Usages* in that case) as their corresponding *Model Definition*.
- *Connections* can be realized between *ports* of *Model Definitions* or *Model Usages*. They represent data flows, or an interface providing/consuming relation, or even any other specific behaviour (by means of *System Interfaces*), according to the nature of the connected *ports*.

By means of *Model Definitions* and *Model Usages*, the SimMF metamodel allows to design simulation models recursively, that is to say building complex models by assembling smaller ones. It also promotes reusability of models, by splitting them in isolated entities (the *Model Definitions*).

The Eclipse EMF project allows generating Java source code from an Ecore metamodel. This source code relies on the EMF runtime framework and permits to manipulate in-memory instances of the metamodel as well as persisting these instances in OMG XMI format. It also automatically deals with cross references between files, thanks to a URI mechanism.

SimMF makes use of EMF to generate the API that permits to manipulate instances of the SimMF metamodel.

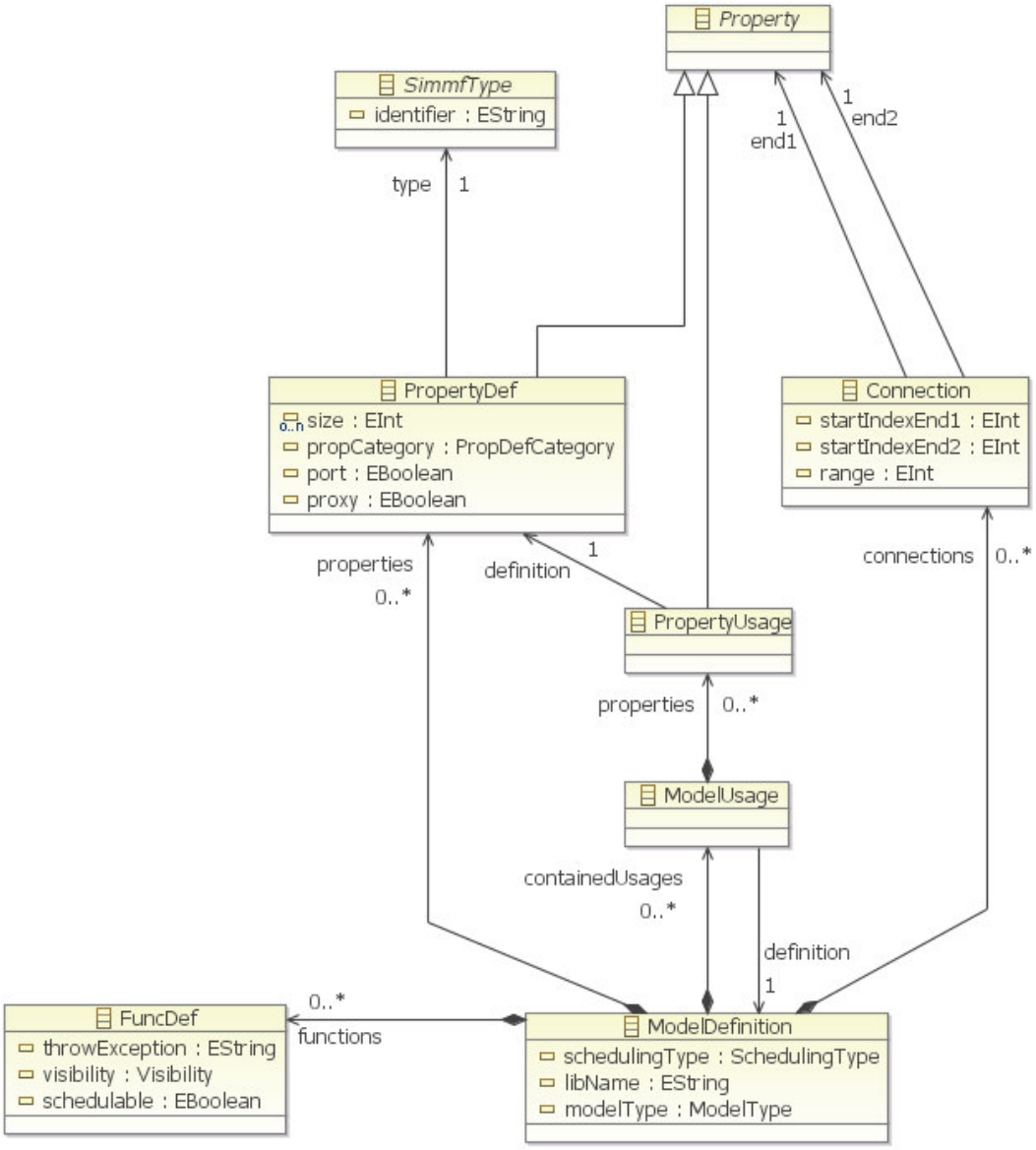

Fig. 3. The core of the SimMF metamodel

INTEGRATED EDITOR

One of the main achievements of SimMF is its integrated editor. The integrated editor is an editor comprised of sub editors. Modifying a data in one of those sub editors causes each other editor to reflect this change, without having to save. Each editor is implemented as a different view, often partial, of the same metamodel. Each one benefits from a generative technology.

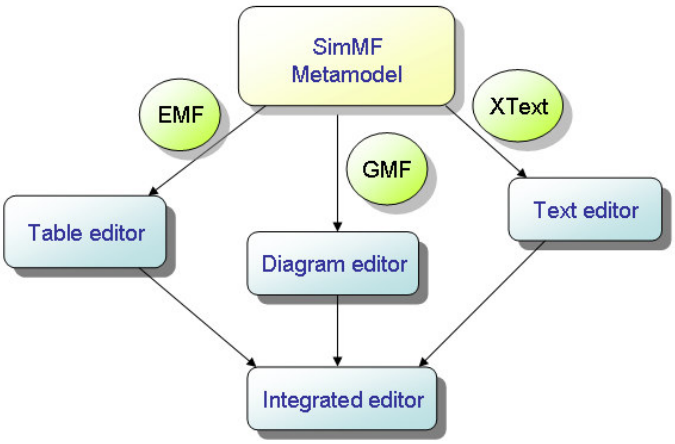

Fig. 4. Usage of EMF, XText and GMF to build the editors of SimMF

Model Overview
The Model Overview shows some basic information of the simulation model. It shows properties like name, description and library name to give a short overview of the opened simulation model.

Model Values
The Model values editor shows all simulation model properties in table based form. All properties can be modified. Following figure shows a screenshot of the model values editor with the model variables tab activated.

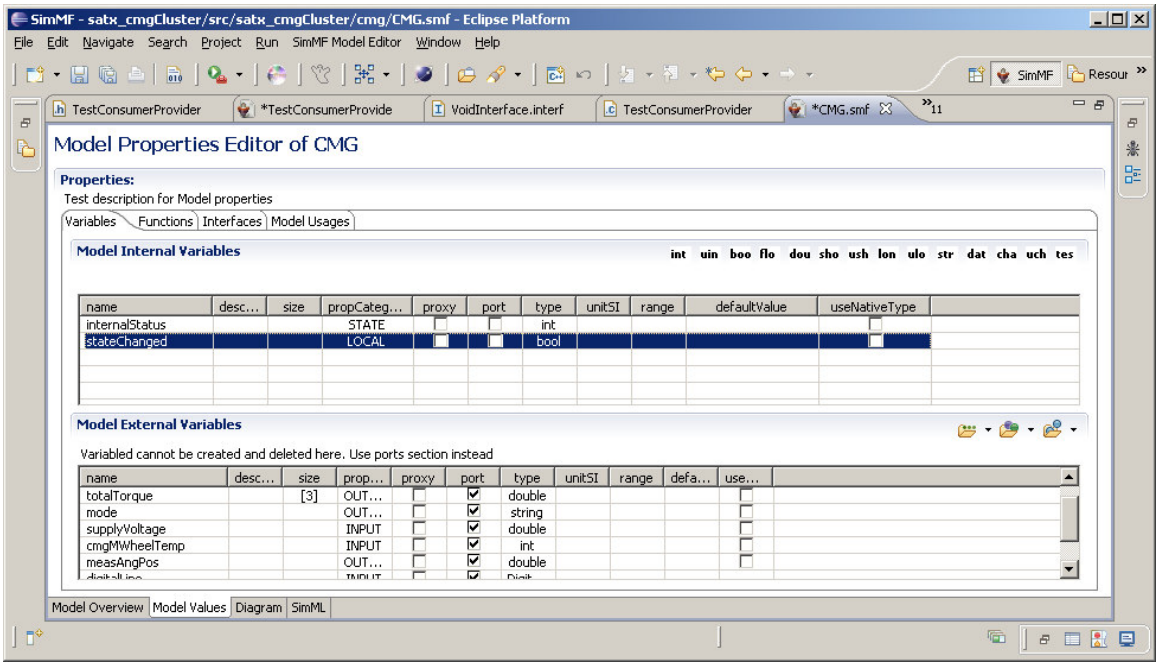

Fig. 5. Model values editor showing variables

SimML Text Editor
The SimML Text editor offers a possibility to modify the model by text. Currently editing of functions is supported only. Function signatures can be entered using the BNF. Afterwards the functions get parsed and added to the simulation model. The SimML Text editor is realized using customized editors from the xText[4] framework. For the editing a temporary text file is created when the integrated editor is opened, containing the function signatures. The editor offers full support for auto-completion and template integration. Following figure shows a screenshot of the SimML text editor.

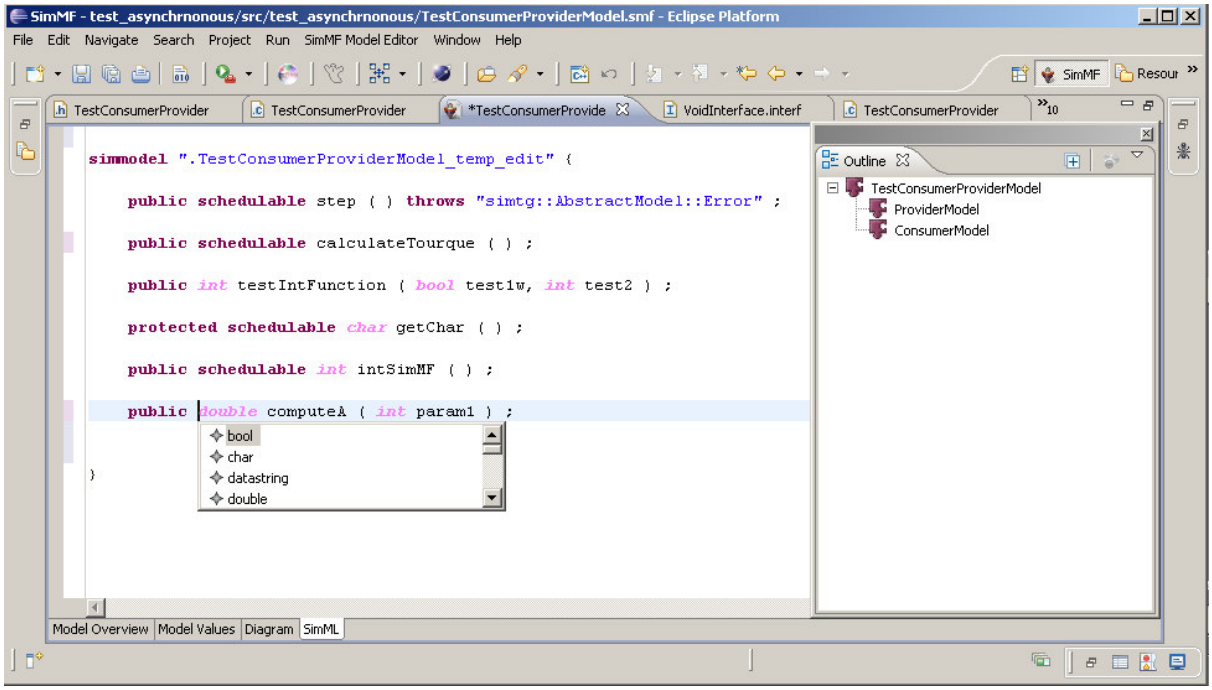

Fig. 6. Text editor for model functions

Diagram Editor

The diagram editor relies on the GMF[5] technology. Its base classes are generated by using the GMF tooling. SimMF provides a set of GMF models that allows generating a base diagram editor.

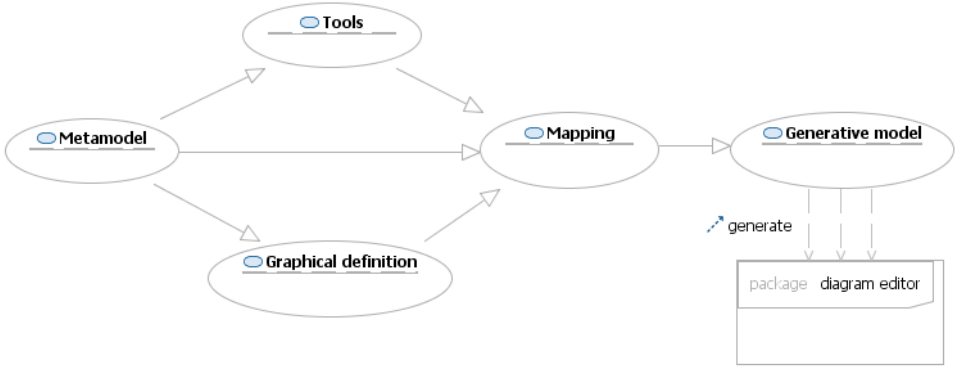

Fig. 7. The models involved in the generation of the base diagram editor

However, most of the classes of the SimMF diagram editor are customized to meet specific SimMF requirements. For instance, SimMF does not make use of the generated tools, but provides a specific mechanism called the *dynamic tooling*. The tools available in the SimMF diagram are computed dynamically, according to the model files and the type files in the workspace. For example, when a model is added in a project, all other models should be able to instantiate it and use it. Therefore a new tool is automatically created in the palette of each editor that is currently open on one of these models. To improve the performance of this palette reloading, the palette factory mechanism of GMF has been replaced. The new one permits to share a set of tools associated to the same project between all editors. When a resource change happens in a project, this unique set then simply has to be rebuilt.

The following figure shows an example of diagram edited by the diagram editor. In this figure, *CMGSensor* is a *Model Definition*, whereas *Tachymeter* and *OpticalEncoder* are *Model Usages* (instances).

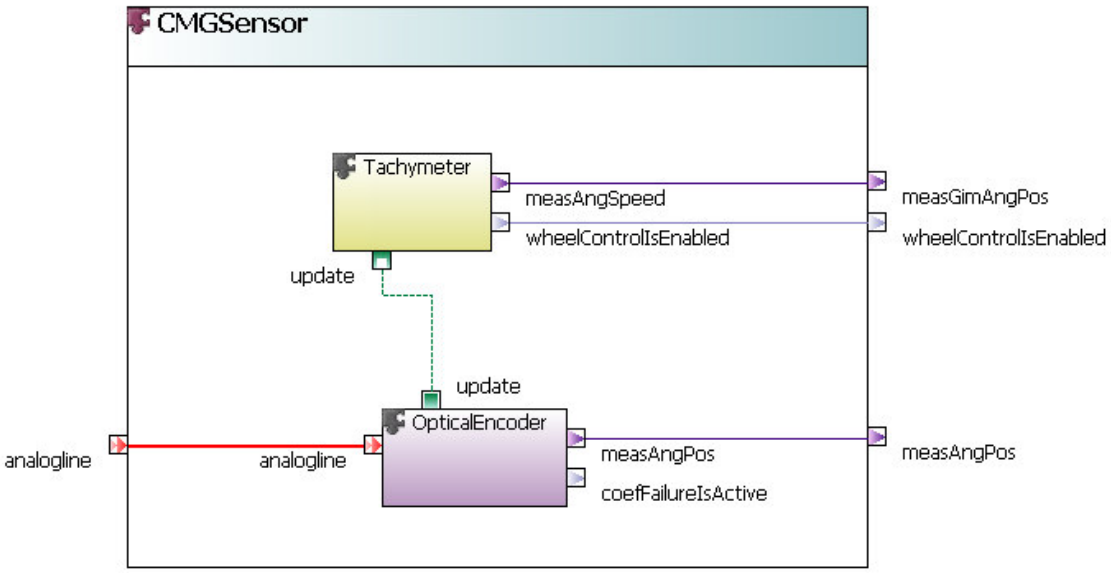

Fig. 8. A diagram in the diagram editor

Among innovative features, the SimMF diagram editor brings improved popup bars: browsable ones (ones that contain folders) and some that serve as quick fix.

Fig. 9. Popup bars brought by SimMF

INCREMENTAL BUILDER

SimMF provides an incremental builder. The incremental builder is triggered by the Eclipse platform when a resource change event occurs in the workspace. The SimMF builder examines this resource change event, and computes the list of files that have to be rebuilt consequently. The list of files to be built is comprised of model files that changed, and of model files that depends on the changed files. For instance, if model A is modified, and if B makes use of A (B has *Model Usages* of A), then both A and B will be rebuilt by the SimMF incremental builder. Such referencing files can also exist outside of the project, by means of referenced project. So the builder also considers the referencing projects of the project that holds the resource change event.

The build process here consists in checking the consistency of the files and generating the code out of them. The major advantage of this builder is that it guarantees smart consistency checks and automatic code generation without any user action. When the user modifies a model and saves it, then he finds instantly the changes implied in the generated code. This is also guaranteed to be done with the minimal effort because of its incremental nature: it builds exclusively what has to be rebuilt, avoiding platform overload.

DATA CONSISTENCY

SimMF features an advanced data consistency checking system. Validation is realized on the model files stored on the disk, but SimMF also provides in-memory consistency checks (or "live checking").
Consistency checks ensure:

- all types of a model are existing and well-referenced
- all connections are valid according to *port* types and directions
- cyclic dependencies detection

Live consistency checks

Live consistency checks inform the user of the in-memory state of his model. They give valuable information about what is valid and what is not, even before the model is saved.
When a model file is open by an editor, the corresponding EMF resource is loaded in memory. Instantly, a Consistency Listener is attached to this resource, and listens to any change. Whenever an interesting change happens, a Consistency Report is created and sent to every registered editor. If an editor is interested by the resource a report has been emitted for, then it reads the report. Thus the editor can determine its state (valid or invalid), and reflect it in the user interface. For instance, in the diagram editor, error markers are displayed on invalid connections. The figure below shows the process of live consistency checking.

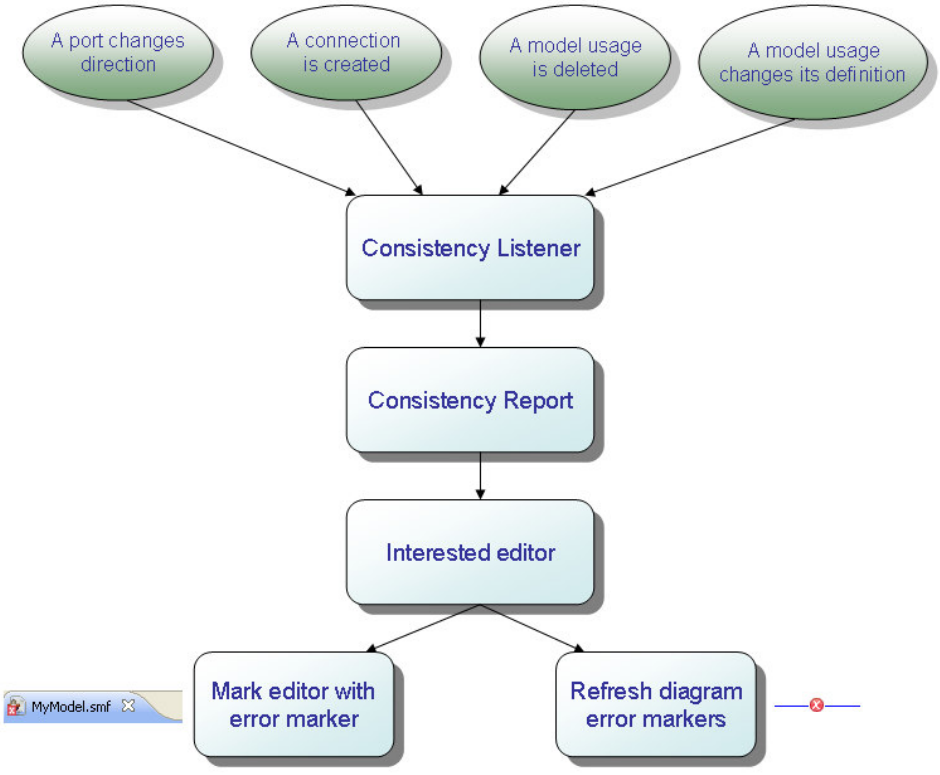

Fig. 10. Live consistency checks

Target specific checks

Those checks are triggered by the incremental builder. They are especially related to a target platform, that is to say a language of which source is generated by SimMF, from SimMF models. For instance, the C++ specific validator checks that the name of the model library does not include any dot (namespace compilation), or that property names do not include some special characters that would cause a compilation error. These specific validators generate warnings, and do not prevent code generation.

SIMMF CODE GENERATION

SimMF generates code which is compatible with the SimTG model ICD. This means that the model can be efficiently integrated on SimTG kernel with other models compatible with this ICD. This ICD is based on SMP2 standard. It is worth mentioning that in contrary to SMP2 approach, the SimMF approach does not distinguish model design phase and model integration phase. These two phases are performed seamlessly for an improved modelling efficiency.

Source code is automatically generated from the models, supporting synchronous and asynchronous modes. It can be customized using merge sections to add algorithmic behaviour.
The C++ generated files for a model A (A.smf) are as following:

- A.hpp – This file is fully generated. It contains methods and fields declaration.
- Abase.cpp – This file is fully generated. It contains all the generated implementation of the class, constructor, destructor, connections and boilerplate code.
- A.cpp – This file is meant to be edited by the user, and contains the implementation of all user defined methods of the *Model Definition*.

The following figure exhibits the code generation process of SimMF. It makes use of a reusable set comprised of a C++ metamodel and the associated model-to-text transformation using the XPAND technology. The business logic of the generation is therefore moved into the transformation engine of SimMF metamodel instances to C++ metamodel instances.

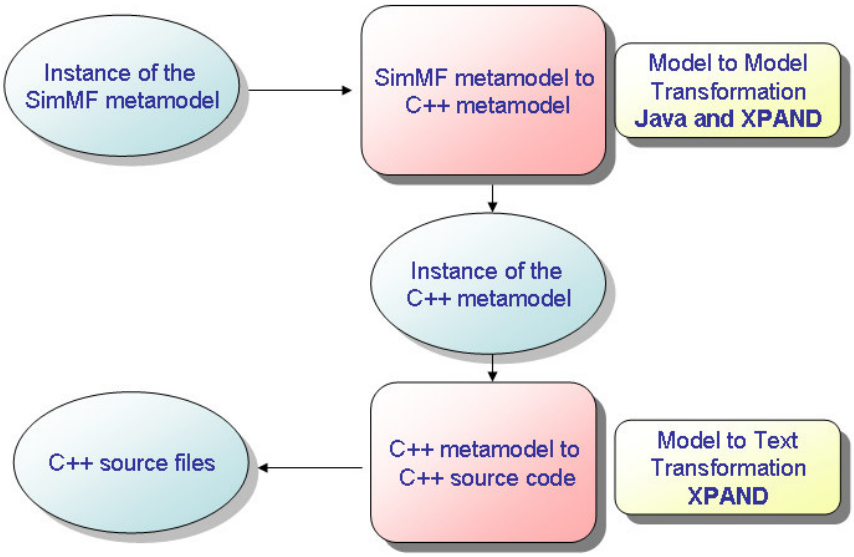

Fig. 11. Code generation process

BUILDING FEATURES

Not only the user can design his simulation models and generate code from them, but he can also build and deliver them within the same environment. The building system relies on the Ant technology, which proved its expressiveness and portability, as well as its ability to address different technologies, like C++ and Java.

When creating a SimMF project, an Ant build script is automatically generated. This Ant script makes use of specific tasks provided by SimMF. Among those tasks, some are responsible for the delivery of SimMF models as protected models – models of which implementation is hidden. Some other are responsible for reading the configuration data stored in the Eclipse project, such as the referenced projects. Those tasks permit to avoid duplication of configuration.
Another task permits to trigger the C++ code generation. This task offers two levels of performance optimizations. The first one is that code generation is not triggered on files that do not need it (by means of time stamps comparisons). The

second one is that source files are not overwritten if the generated code is rigorously similar to the previous one (by generating code in a buffer and comparing it with the file), avoiding timestamp modification and thus saving useless recompilation.

Thanks to those Ant tasks, the simple script generated at the project creation permits to build an average SimMF project out-of-the-box. It deals automatically with referenced project at source level but also with referenced project that have been delivered outside of the workspace (e.g. anywhere else of the local network). Moreover, this build works on both Linux and Windows platforms.

PRODUCT ASSURANCE

SimMF benefits from automatic user interface testing. To do so, it makes use of the SWTBot[6] technology. SWTBot allows writing automated graphical user interface tests. It brings automation of any user actions, like clicking on a menu or dragging a figure in the diagram editor. What is more, tests are integrated with a continuous integration tool, namely Hudson.

Whenever a SimMF developer commits a modification of the SimMF code base, a build and integration test of SimMF is automatically triggered by Hudson[7] and all user interface tests are ran consequently. Then test reports are automatically collected as well as screenshots of UI tests that failed (see figure below).

The whole system gives SimMF the best of continuous integration and automated testing, and ensures an excellent quality level of the product.

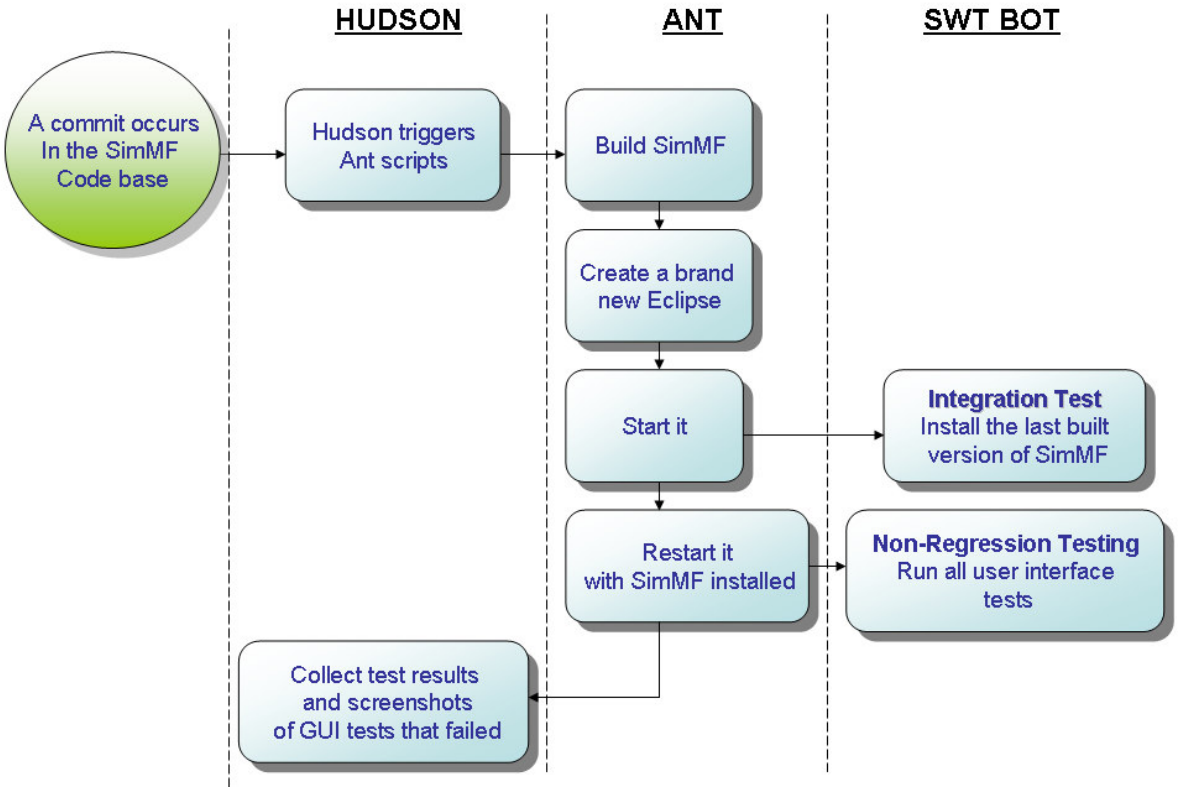

Fig.12. Continuous integration and non-regression testing

REFERENCES

[1] OMG MOF specifications. http://www.omg.org/mof/
[2] Eclipse project homepage. http://www.eclipse.org/
[3] EMFproject homepage. http://www.eclipse.org/modeling/emf/
[4] XText project homepage. http://www.eclipse.org/Xtext/
[5] GMF project homepage. http://www.eclipse.org/modeling/gmp/
[6] SWTBot project homepage. http://www.eclipse.org/swtbot/
[7] Hudson project homepage. http://hudson-ci.org/