Dynamic Interface for Multi-physics Simulators

César Oliveira[1], Fernando Rocha[1], Renata Medeiros[1], Ricardo Lima[1], Sérgio Soares[1], Félix Santos[2], and Ismael H. F. Santos[2]

[1]Department of Systems and Computing
University of Pernambuco

[2]Federal University of Pernambuco
Department of Mechanical Engineering
Rua Acadêmico Hélio Ramos, s/n - Recife - PE 50740-530 - Brazil

[2]CENPES Petrobrás, Rio de Janeiro, RJ

{cesar.lins, rfernandoafr, renatawm}@gmail.com, {ricardo, sergio}@dsc.upe.br, flxcgs@yahoo.com.br, ismaelh@petrobras.com.br

***Abstract*—Simulation is a well known technique to study complex systems. However, the implementation of a simulator may be more complex than the simulation itself. For instance, graphical user interface (GUI) development might consume around 50% of the software development time. Therefore, strategies and techniques to reduce costs in the development of GUI are mandatory in modern software engineering. In this paper we present a framework called GUI Generation Tool that dynamically constructs user interfaces based on specifications defined in XML files. This framework was defined to support automatic generation of simulators for multi-physics phenomena using software reuse techniques and software product lines concepts.**

***Index Terms*—GUI Generation, Dynamic Interface**

I. INTRODUCTION

Two main concerns are common in complex systems development: 1) reducing costs; 2) improving safety. Hence, designers must analyze the system to understand different aspects involved in its implementation. Simulation is a well known technique to study complex systems. However, in some cases the implementation of the simulator may be more complex than the simulation itself. Thus, defining strategies and techniques to automate as much as possible the generation of simulators is very important.

MPhyScaS (Multi-Physics Multi-Scale Solver Environment) is a work in progress to define an environment for automating the development of simulators based on the finite element method. The term multi-physics can be defined as a qualifier for a set of interacting phenomena, in space and time. These phenomena represent deformation of solids, heat transfer, electromagnetic fields, etc.

MPhyScaS adopts software reuse techniques and software product lines concepts to partially automate the generation of multi-physics simulators. In particular, MPhyScaS will reuse tested software components to assemble the core of multi-physics simulators. It is possible that components have the same functionality, however, executing in a different way. This allows better flexibility, since the user can select which components to use, modifying the characteristics of the simulation. Such components might require different user data and therefore their own graphical user interface.

In order to better support the automatic generation of different simulators, also increasing development productivity, we present a framework for dynamically generating user interfaces for simulators generated by MPhyScaS. The framework is called GUI Generation Tool.

The structure of this paper is as follows. Section II presents MPhyScas and its architecture. The software product line concept is presented in Section III. Section IV presents the automatic graphical user interface generation tool and Section V describes the dynamical interface for the MPhyScas simulators. Finally, S ections VI and VII present related work and conclusions.

II. MPHYSCAS

MPhyScas (Multi-Physics Multi-Scale Solver Environment) is an environment dedicated to the automatic development of simulators based on the finite element method. The term multi-physics can be defined as a qualifier for a set of interacting phenomena, in space and time. These phenomena are usually of different natures (deformation of solids, heat transfer, electromagnetic fields, etc.) and may be defined in different scales of behavior (macro and micro mechanical behavior of materials). A multi-physics system is also called a system of coupled phenomena. If two phenomena are coupled, it means that part of one phenomenon's data depends on information from other phenomenon. Such a dependence may occur in any geometric part, where both phenomena are defined. Other type of data dependence is the case where two or more phenomena are defined on the same geometric component and share the geometric mesh. Multi-physics and multi-scale problems are difficult to simulate and the building of simulators for them tend to be very demanding in terms of time spent in the programming of the code. The main reason is the lack of reusability. A detailed discussion can be found in (colocar Referencia).

Usually, simulators based on the finite element method can be cast in an architecture of layers. In the top layer global iterative loops (for time stepping, model adaptation and articulation of several blocks of solution algorithms) can be found. This corresponds to the overall scenery of the simulation. The second layer contains what is called the solution algorithms. Each solution algorithm dictates the way linear systems are built and solved. It also defines the type of all operations involving matrices, vectors and scalars, and the moment when they have to be performed. The third layer contains the solvers for linear systems and all the machinery for operating with matrices and vectors. This layer is the place where all global matrices, vectors and scalars are located. The last layer is the phenomenon layer, which is responsible for computing local matrices and vectors at the finite element level and assembling them into global data structures.

The definition of those layers is important in the sense of software modularization. But it does not indicate neither how entities belonging to different layers interact nor what data they share or depend upon. That is certainly very important for the definition of abstractions, which could standardize the way those layers behave and interact. The architecture of MPhyScas presents a language of patterns in order to define and represent not only a set of entities in each layer - providing the needed layer functionalities - but also the transfer of data and services between the layers. Thus, MPhyScas is a framework that binds together a number of computational entities, which were defined based on that language of patterns, forming a simulator. Such a simulator can easily be reconfigured in order to change solution methods or other types of behavior (colocar referencias sobre MPhyScas). Almost every single piece of code that constitute MPhyScas computational entities in a simulator can be reused in the building of other different simulators. This makes the simulators produced by MPhyScas strongly flexible, adaptable and maintainable.

A. *MPhyScaS's Architecture*

The architecture of MPhyScas-S was proposed in [1]. This architecture establishes a computational representation for the computational layers using patterns (see Figure 1), where, the **Kernel** Level represents the global scenery level, the level of the solution algorithms is represented by the **Block** Level, the level of solvers is represented by the **Group** Level and the phenomena level is represented by the **Phenomenon** Level. The definition of this structure is aimed at improving the quality of simulators designs. The defined architecture attempts to fill in the existing gap in the development of FEM simulators for multi-physics and multi-scales problems. The main requirements of this architecture are:

- Flexibility in the development of simulators;
- Extensibility of simulators through the integration of components;
- Improved reusability of processes, data and models.

The architecture of MPhyScas is shown in Figure 2. The Static Library allows the maintenance of data employed in the building of simulators and simulations. Those data includes: methods (mesh generation, numerical integration, for

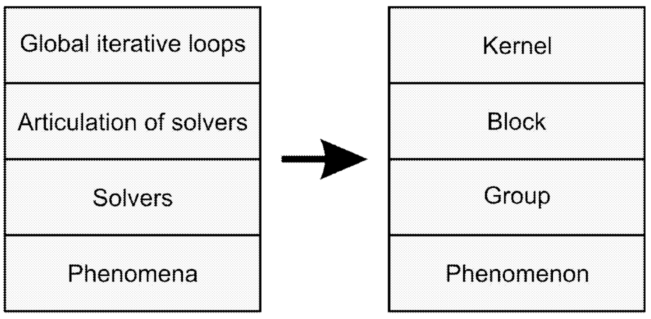


Fig. 1. Computational representation for the layers of the simulator

instance), functions (constitutive parameters, for instance), algorithms and phenomena. The Pre-Processor produces **Data for Simulation** and builds the **Simulator** using the **Static Library**. The **Data for Simulation** represent the input data in a simulation, which are used by the **Simulator**. The **Simulator** is responsible for the execution of a simulation. The **Simulator** uses the **Data for Simulation** and produces the **Results of the Simulation**. The **Viewer** uses the **Results of the Simulation** and the **Data for Simulation** to produce the visualization of the simulation results.

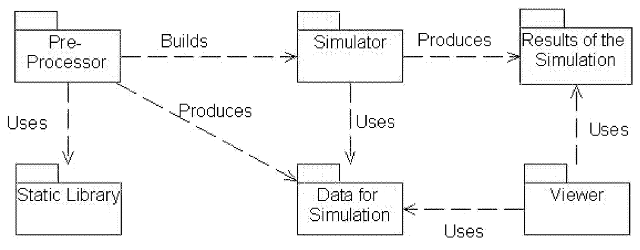


Fig. 2. Architecture of the MPhyScas

The **Simulator** is structured as a tree, divided into four layers:

- **Kernel**: it is responsible for initialization procedures (transferred to **Blocks** in the lower level); for global time loops and iterations; for global adaptive iterations and for articulation of activities to be executed by the **Block**s in the lower level. The **Kernel** stores system data related to the parameters for its loops and iterations;
- **Block**: it is responsible for the transfer of incoming demands from the **Kernel** to its **Group**s in the lower level (initialization procedures, for instance); for **Block** local time loops and iterations (inner loops and iterations inside a global time step, restricted to groups of phenomena); for procedures inside time stepping schemes; for **Block** local iterations (restricted to some groups of phenomena, like in a Newton-Raphson iteration, for instance); for **Block** local adaptive iterations (restricted to some groups of phenomena); for operations with global quantities (transferred to **Group**s in the lower level, which are the owners of global quantities). The **Block**s serve the **Kernel** level. Each **Block** is responsible for a certain number of **Group**s, which can not be owned by other **Block**. All demands from a **Block** to the lower level should be addressed to its **Group**s. The **Block**s store system data

related to parameters for their own loops and iterations and parameters for their procedures;

- **Group**: it is responsible for the transfer of incoming demands from its **Block** to **Phenomena** in the lower level (initialization procedures, for instance); for the assembling coordination and solution of systems of linear algebraic equations (the method used depends on the solver component); for operations with global quantities (by demand from its **Block**), for articulation of activities to be executed by its **Phenomena** in the lower level (basically concerned with computation and assembling of global matrices and vectors). The **Group**s serve their respective **Block**s. Each **Group** is responsible for a certain number of **Phenomena**, which can not be owned by other **Group**. All demands from a **Group** to the lower level should be addressed to its **Phenomena** only. The **Group**s store global matrices, vectors and scalars and store the **GroupTask**s, which are objects encapsulating standard procedures, where articulation of the **Group**'s **Phenomena** are needed. The **GroupTask**s are programmable and their data are standard pieces of information, depending only on the type of the **GroupTask**.
- **Phenomenon**: it is responsible for the computation of local matrices, vectors and scalars (**Phenomenon** quantities); for operations involving matrices and vectors at the finite element level and their assembling into given global matrices and vectors. The **Phenomena** serve their respective **Group**s. The **Phenomena** store data related to constitutive parameters or other parameters, which are specific of the respective Phenomenon; store the geometry where the respective **Phenomenon** is defined (different **Phenomena** may share a geometry or a part of it); store **WeakForm**s, which are tools for computing and assembling quantities defined on a certain part of the geometry. A **WeakForm** may be active or not. Only active **WeakForm**s can be used during a simulation. A **WeakForm** may store parameters, which are related to specific simulation data (for instance, functions for the definition of boundary conditions or parameters needed for the computation of a quantity, which should be given together within a simulation data set). The **Phenomenon** should store methods, which are tools to be used in certain **Phenomenon** specific tasks. For instance, those tasks can be generation of geometric and **Phenomenon** meshes, numerical integration at the element level, shape functions, etc.

The simulation starts with the execution of the root of the **Kernel**, which uses services provided by a set of **Block**s, which in turn uses services from a set of **Group**s. Each **Group** owns a set of **Phenomenon** objects, which are used to perform the production of local matrices and vectors and the assembling of them into given (by the **Group**) global matrices and vectors. Observe that the **Kernel** may articulate several **Block**s; each **Block** may articulate several **Group**s; each **Group** may articulate several **Phenomenon**s and each **Phenomenon** is able of computing an assembling several local quantities (scalars, vectors or matrices), as it can be seen in Figure 4.

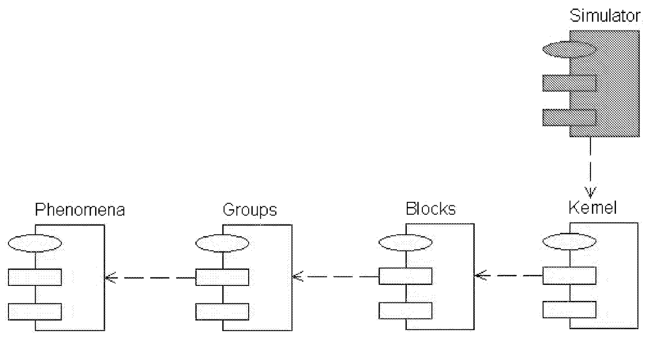


Fig. 3. Simulator diagram

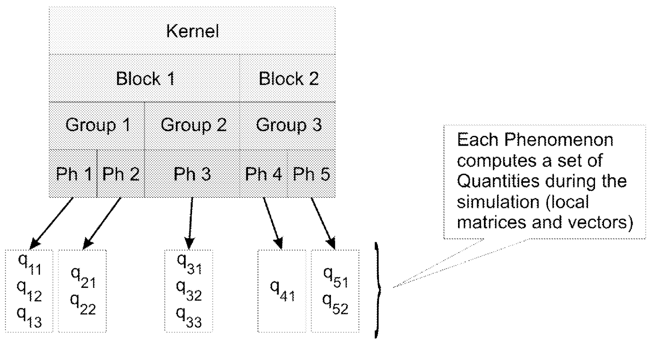


Fig. 4. Each Phenomenon object is able of computing a set of quantities during a simulation

The states that define the configuration of each **Phenomenon** object are stored in the respective **Group** object, where solvers are located. This is convenient due to the fact that the **Group**'s layer is responsible not only to assemble and solve algebraic systems, but also to operate with scalars, vectors and matrices in response to requests from a **Block** (see Figure 5.

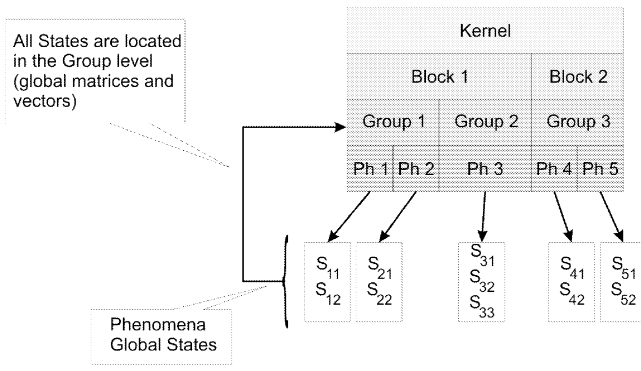


Fig. 5. Each Phenomenon object has its own set of states, which is stored in its Group object

A quantity that a **Phenomenon** object can compute and assemble may be coupled to other **Phenomenon**'s states (one or more) as it is depicted in 6. MPhyScas-S provides all the machinery to make this procedure automatized following the specification of some data related to the place where coupling occur, handlers for the states and a reference to the coupled **Phenomenon** object, which should be given to the object responsible for the computation.

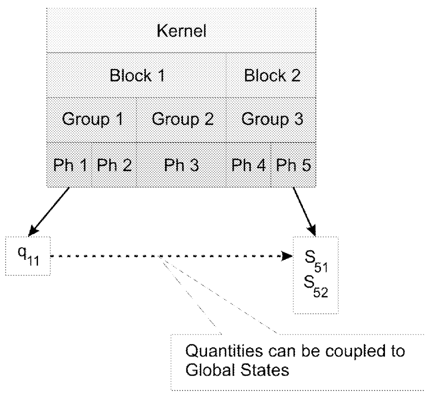


Fig. 6. A quantity computed by a Phenomenon object may be coupled to a state from other Phenomenon object

For further detailed information on MPhyScas see [2].

III. Software Product Lines

Software reuse is a debated topic in Software Engineering research. This technique attempts to reduce redundant effort by reusing components that were developed for other similar products [3]. To take relevant advantages from this idea, the development must be planned and executed in a way that emphasizes a reuse politics.

In this connection, the concept of *Software Product Lines* (SPL) [4] was introduced. It defines an environment in which systems with similar characteristics are produced by assembling *assets* previously developed. The idea is based in the fact that few products are really unique. Anything new that is necessary for a product is created under a *reuse policy* and can be reused in the future for assembling other products.

An SPL can be described by four basic concepts: (**i**) **reusable software assets management**, which identifies and registers reusable assets produced during the development. Examples of assets are requirements, software components, test cases and documentation; (**ii**) **model decision**, which defines the structure of the final product based in similarities and variation points; (**iii**) **production mechanism**, which is responsible for supporting the assembling of software assets, using the model decision, and for generating the output product; and (**iv**) **product output**, which is the set of final products of the SPL. Figure 7 illustrates this process.

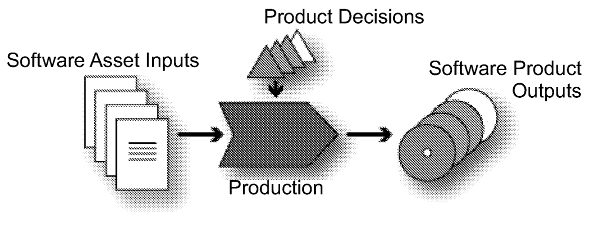


Fig. 7. Process of a Software Product Line

According to Griss [5], an SPL is a group of products that share some common items, but also has some significant variations. The main goal of an SPL is reducing costs related to the development and maintenance of common domain software. In an scientific simulation domain, many elements are repetitive, like mathematical methods and structures. New algorithms that are designed for one simulator are likely to be useful for other related simulators. In this context, a reuse politics is of great value.

The MPhyScaS environment supports the SPL approach to minimize the development and maintenance costs of multi-physics simulators. It defines a modular architecture to which scientific components can be connected for creating different purpose simulators. In addition, it provides a repository of components and a tool for automatically assembling the final products.

IV. GUI Generation

The definition of Graphical User Interfaces (GUI) is an important aspect of software quality. It has a direct effect on the user productivity.

Usually, the GUI consumes in average 50% of the system development time [6]. Thus, the adoption of strategies and techniques to reduce costs in the development of GUI is mandatory in modern software engineering.

In a Software Product Line (SPL) [4] environment, this effort needs to be addressed with great care. Each component integrated in an SPL may need a specific GUI. However, implementing every interface from the scratch has a significant impact in the cost of products.

In an SPL environment, the similarity between the products allows reusing solutions already developed for other products. When implementing for SPL, attention must be given to the commonness and the differences between each product. The differences are identified as variation points.

A. GUI Generation Tool

The approach proposed by Liborio et al. [7] defines three phases to automatically generate GUI. In the first phase, the interface definition is guided by the tasks that users will perform through the GUI. In the second phase, the user task models are translated into an abstract representation of the GUI. The final phase transforms the abstract model into the concrete GUI.

This work proposes a GUI generator for MPhyScaS [8]. The generator is implemented as a framework called GUI Generation Tool. It is divided in two modules: the *Creator* and the *Builder*.

We adopt the approach defined by Liborio et al. Hence, the tasks models are represented by the set of parameters that the user must provide to each component. These parameters are defined in a graphical interface that corresponds to the *Creator* module. They are then mapped into an XML (eXtensible Markup Language) representation. After that, the XML is read by the *Builder* module, which constructs the interface. This process occurs in two steps: *component registration* and *interface instantiation*. Each component in the MPhyScaS repository must be registered before its inclusion into a simulator. During registration, the GUI Creator interface is displayed to the user for defining its parameters. After that, an XML file with this definition is created. This corresponds to the first step.

An XML file is generated for each component in the simulator. GUI Builder manages the dependence relation between components. Thus, if the user accesses a feature that requires a component, the GUI Builder reads the corresponding XML file and generates the interface to configure the component parameters. This corresponds to the second step.

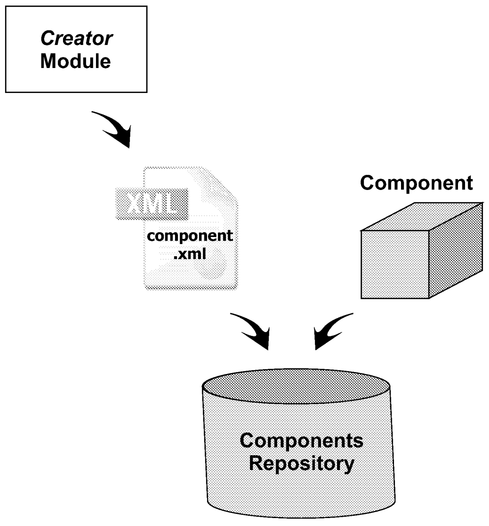


Fig. 8. Architecture of Module *Creator*

Easy integration with other system is an important requirement of the GUI Generator tool. In particular, its integration with the MPhyScaS can be observed in two points: the simulator generator uses the *Creator* module; while the *Builder* module composes all the simulators generated by MPhyScaS. The architecture of both modules is illustrated in Figures 8 and 9.

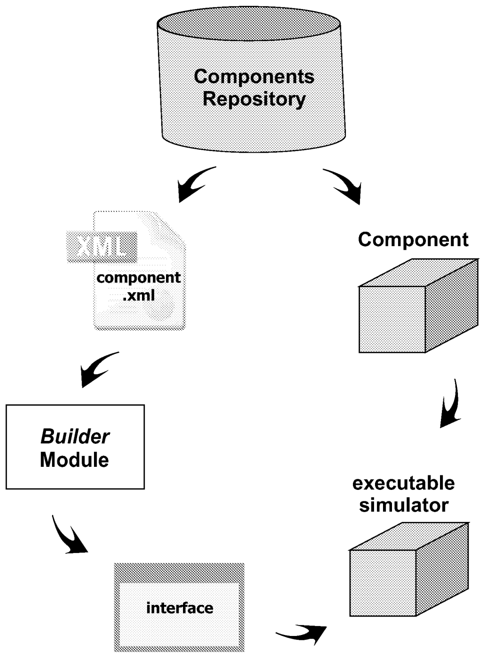


Fig. 9. Architecture of Module *Builder*

The Builder also validates the values inserted by the user. The set of valid data are described through value constraints in the XML description. The tool presents an error message if the user inserts an invalid value.

When a parameter value is changed by the user, the framework notifies the application, allowing it to update the models accordingly.

TABLE I
CONSTRAINTS OF THE GUI GENERATOR

Type	Implication
NUMBER	The value is a number
POSITIVE	The value allows positive numbers (needs the number constraint)
NEGATIVE	The value allows negative numbers (needs the number constraint)
ZERO	The value allows the zero value (needs the number constraint)
MAX_VALUE	The value has to be less than the value of this constraint (needs the number constraint)
MIN_VALUE	The value has to be greater than the value of this constraint (needs the number constraint)
REAL	The value is a Real number (needs the number constraint)
INTEGER	The value is a Integer number (needs the number constraint)
STRING	The value is a text
EMPTY	The parameter allows empty value

B. Format of the Component Description File

The XML file used to describe component parameters is structured as shown in Figure 10.

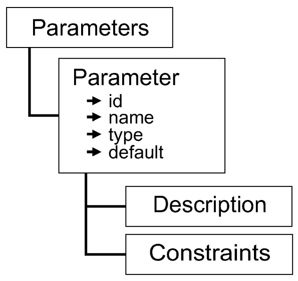


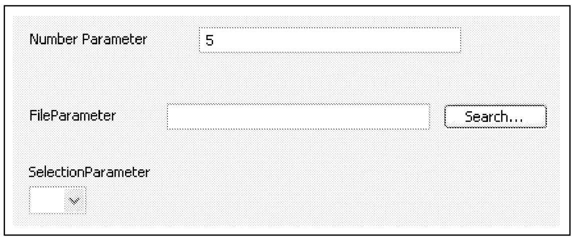
Fig. 10. Component structure

The root element is the `Parameters` structure. It contains a set of `Parameter` elements. Parameters are identified by an attribute *id* and a *name*. This name is intended to be presented to the user in the generated interface. The attribute *type* is required to classify the GUI according to the following class of interface:

- Selection: the user must choose a value from a list of values;
- Checklist: the user can chose many options in a list;
- File: a file name and path must be provided by the user;
- Prioritized List: the user has a list of values and can change the order of the items;
- Check Box:a conventional *yes* or *no* check box;
- Text: any textual parameter, including numbers.

Optionally, a default value can be assigned to the parameter by setting the attribute `default`.

The `Parameter` contains an element `Description`, to provide a textual description and an element `Constraints`.

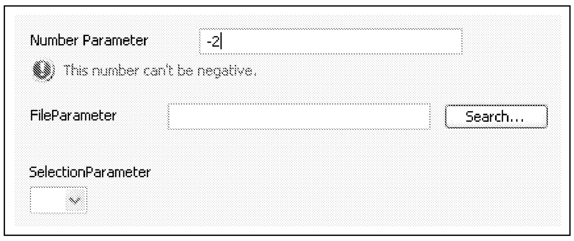


Number Parameter 5

FileParameter Search...

SelectionParameter

Fig. 11. Example of generated interface



Number Parameter -2

This number can't be negative.

FileParameter Search...

SelectionParameter

Fig. 12. Example of error message displayed by the generated GUI

Such `Constraints` are required to validate the data entered by the user in the interface. Examples of constraints are the range within a numeric parameter, the accepted file extension, or the size of a text parameter.

The constraints supported by the GUI Generator are described in Table I.

C. Example of Generated GUI

The interface is generated by instantiating SWT widgets according to the parameter description. Figure 11 shows an interface created through the GUI Generator Tool. Such an interface corresponds to the GUI defined by the XML presented bellow. The example illustrates three types of parameters: Text (with number constraint), File, and Selection.

When the user inputs an invalid value into a data filed, the interface displayes an error message. This is exemplified in Figure 12.

```
<?xml version=”1.0” encoding=”UTF−8”?>
<parameters>
  <param id=”0” name=”Number␣Parameter” type=”NUMBER
      ” default=”5” description=””>
    <constraints>
      <positiveConstraint />
      <integerConstraint />
    </constraints>
  </param>
  <param id=”1” name=”FileParameter” type=”FILE”
      default=”MS␣Word␣$␣∗.doc” description=””>
    <constraints />
  </param>
  <param id=”2” name=”SelectionParameter” type=”
      SELECTION” default=”” description=””>
    <constraints>
      <selectionConstraint value=”v1␣#␣v2␣#␣v3” />
    </constraints>
  </param>
</parameters>
```

V. MPhyScaS Dynamic Interfaces

The MPhyScaS simulation interface supports a great variety of simulators for many purposes. These simulators share the same layer structure presented in Section IV, but may differ regarding the number and types of phenomenon, solution algorithm, numeric methods, mesh generators and others. Thus, the data manipulated by each simulator is usually different. Therefore, creating a new interface for every new simulator designed would demand considerable effort.

Following the Software Product Line approach, the MPhyScaS offers a generic interface. Such an interface is a framework to which components can be integrated. Thus, in order to construct the simulator, the user simply connect the desired components to the generic interface. Then, the framework reads the XML description of each component and assemble the simulator interface.

The framework includes facilities to: 1) visualize the simulator structure the components available; 2) configure each component; 3) transfer the data entered by the user to the executable simulator. This is sketched in Figure 13.

The structure of the simulator is given by an XML description, stored in the file `simulator.xml`. This file describes which components are present in each layer of the simulator structure and how they interact with each other. For example, phenomena and groups in the simulator and how these phenomena are distributed across the groups.

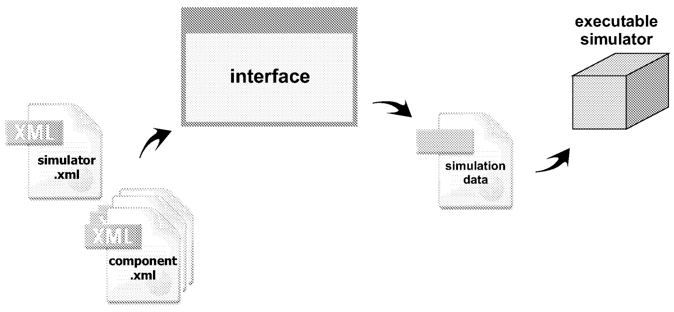


Fig. 13. MPhyScaS interface architecture

Each component of the simulator demands some parameters from the user. Once the interface is informed about the components in the simulator, it requests the parameters required for each component. The parameters definition is stored in a set of XML files. Each component has its own description file. This is represented in Figure 13 by the set of `component.xml` files. Actually, each component will have a description file with a different name. The name of the component's file is provided by the `simulator.xml`.

The interface then can use the GUI Generator to dynamically construct and present the data forms that the components require.

Finally, the information provided by the user is stored in the `configuration.xml` file. This file is constructed according to the definition present in the `simulator.xml` and the components description files. It contains component parameter values, geometry description, quantity activations and so on.

A. Integration of the GUI Generator Framework

Every component in the repository shares the parameter structure described in Sec. IV.

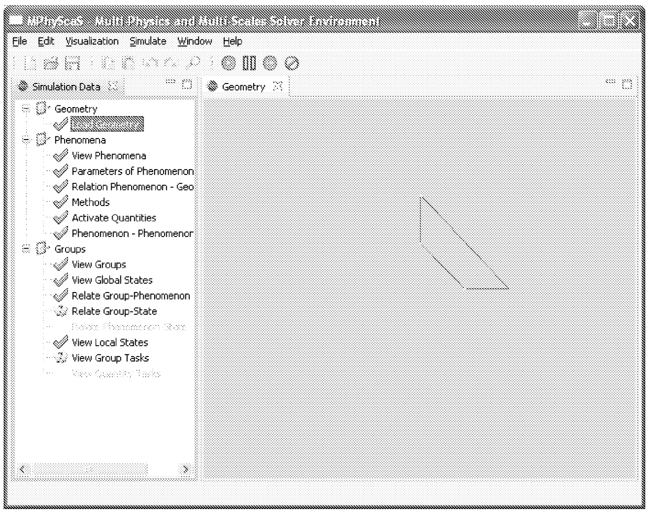
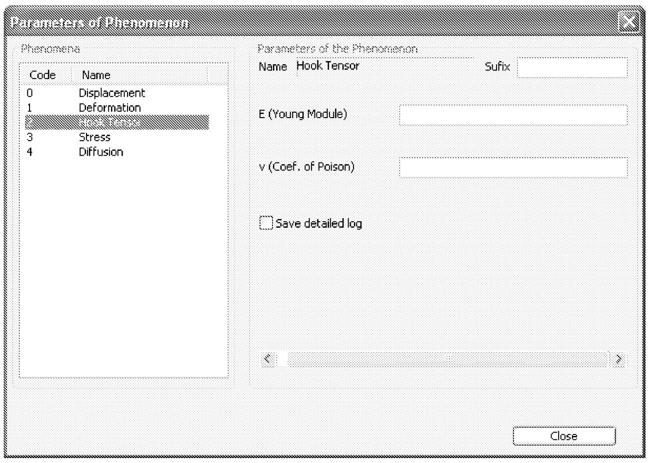


Fig. 14. MPhyScaS interface main window



Code	Name
0	Displacement
1	Deformation
2	Hook Tensor
3	Stress
4	Diffusion

Name Hook Tensor Sufix

E (Young Module)

v (Coef. of Poison)

☐ Save detailed log

Fig. 15. Example of dynamically created screen

The interface uses the information in the `simulator.xml` file to present present the simulator structure (see Figure 14).

The user can navigate through the Simulation Data (left-hand side of Figure 14) by clicking in the desired element. Each such an element corresponds to a *screen* with special forms used to configure the simulation.

Usually, the user accesses a screen which requires some component information. For instance, when the user wants to configure a phenomenon. In these situations, the interface reads the `simulator.xml` and recover the file name which describes such a component. Eventually, it requests the "Builder" to construct the interface for that component.

The *Builder* instantiates widgets for each parameter and returns the corresponding form to the interface. When the user changes a parameter value, the Builder checks if the constraint is satisfied. It rises an exception in case of invalid values. If a valid value is given, the Builder notifies the interface that the parameter has changed and provide the input values as Objects. The interface stores these Objects in Hash Tables. The parameter *id* is as a key for these hash tables.

The Builder is executed once for each component. The resulting interface is maintained in memory and accessed by users whenever they want to configure a component.

Figure 15 shows an example of a form dynamically generated through the GUI Generator Tool. This interface is employed to configure phenomena parameters. When the user chooses a phenomenon in the left-hand side, the Builder constructs the parameter form for that component which is displayed in the right-hand side area of the interface.

The XML description for the phenomenon shown in Figure 15 is presented in the listing bellow.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<parameters>
  <param id="0" name="E␣(Young␣Module)" type="NUMBER
      " default="" description="">
    <constraints>
      <numberConstraint />
      <negativeConstraint />
      <zeroConstraint />
      <positiveConstraint />
      <realConstraint value="4" />
    </constraints>
  </param>
  <param id="1" name="v␣(Coef.␣of␣Poison)" type="
      NUMBER" default="" description="">
    <constraints>
      <numberConstraint />
      <negativeConstraint />
      <zeroConstraint />
      <positiveConstraint />
      <realConstraint value="4" />
    </constraints>
  </param>
  <param id="2" name="Save␣detailed␣log" type="CHECK
      " default="FALSE" description="">
    <constraints />
  </param>
</parameters>
```

The *Bulider* is responsible for creating the XML description of the values entered by users. In order to create the XML, the *Builder* requests the parameter type, the value constraints, and the object stored in the Hash Table. An example of parameter values XML generated by the Builder is displayed in the next listing.

```xml
<parameters>
  <parameter id="0">
    <listData>
      <data value="3.000" />
    </listData>
  </parameter>
  <parameter id="1">
    <listData>
      <data value="−1.000" />
    </listData>
  </parameter>
  <parameter id="2">
    <listData>
      <data value="true" />
    </listData>
  </parameter>
</parameters>
```

VI. Related Work

Several researches have demonstrated how user interfaces can be automatically generated. Some of these researches use declarative modeling language to create a model that represents the interface. The tool proposed by Browne [9] synthesizes this model into run-time code. Costa-Neto [10] proposed a system that generates multi-platform interfaces for the web.

Nichols [11] combines the declarative model into a Rich Human-Agent Interaction (RHAI) technique. Intelligent agents interact to the user for modeling questions that modifies the XML to decide the best way to present the GUI.

The correctness of GUI is an important research area. In particular, Shiffman proposed UIVerify [12], which is able to generate and verify the correctness of web interfaces.

Similar to other works, our approach adopts a declarative language to define the interface model. We decided to use the XML language to represent the interface. We provide users with a visual environment to model the GUI. Such an environment generates the XML model. Hence, no previous knowledge about the MPhyScaS XML model is required to create the interface model.

VII. Conclusions

In this paper we presented a dynamic interface framework, called GUI Generation Tool, to support the automatic generation of different multi-physics simulators. The simulator generator, MPhyScaS, is an environment for automating the development of simulators using reusable components. These components might need different input data, which demands this kind of dynamic infrastructure.

In fact, the GUI Generation Tool can be reused in other environments. The framework gets an XML file as input and based on the file definition creates a GUI to enter the input data. MPhyScaS will generate the XML file for a specific simulator based on the selected components and the used definitionos.

Acknowledgment

This work was supported in part by Finep and Cenpes/Petrobras.

References

[1] LENCASTRE, M. *Conceptualisation of an Environment for the Development of FEM Simulators*. Tese (Doutorado em Ciências da Computação) — Universidade Federal de Pernambuco, Recife, Pernambuco, 2004.

[2] SANTOS, F. et al. Towards the automatic development of simulators for multi-physics problems. *International Journal of Modelling and Simulation for the Petroleum Industry*, v. 1, n. 1, 2007.

[3] MILI, H. et al. *Reuse-Based Software Engineering: Techniques, Organization, and Controls*. [S.l.]: John Wiley and Sons, 2001. ISBN 0-471-39819-5.

[4] CLEMENTS, P. C.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. [S.l.]: Addison-Wesley, 2001. (SEI Series in Software Engineering).

[5] GRISS, M. L. Product-line architectures. In: HEINEMAN, G. T.; COUNCILL, W. T. (Ed.). *Component-Based Software Engineering: Putting the Pieces Together*. [S.l.]: Addison-Wesley, 2001. ISBN 0-201-70485-4.

[6] MYERS et al. Survey on user interface programming. In: *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*. [s.n.], 1992. (Tools and Techniques), p. 195–202. Disponível em: <http://www.acm.org/pubs/articles/proceedings/chi/142750/p195-myers/p195-myers.pdf>.

[7] LIBÓRIO, A. et al. Interface design through knowledge-based systems: an approach centered on explanations from problem-solving models. In: SIKORSKI, M. (Ed.). *TAMODIA*. ACM, 2005. p. 127–134. ISBN 1-59593-220-8. Disponível em: <http://doi.acm.org/10.1145/1122935.1122961>.

[8] ROCHA, F. A. F.; SOARES, S. C. B. Configuração dinâmica de interface com o usuário. 2007.

[9] BROWNE, T. et al. *The MASTERMIND User Interface Generation Project*. [S.l.], 1996.

[10] NETO, M. C.; LEITE, J. Uma proposta para o desenvolvimento de interfaces de usuário multi-plataforma com tecnologia web. In: *IHC 2004 - VI Simpósio sobre Fatores Humanos em Sistemas Computacionais*. [S.l.: s.n.], 2004. p. 235–238.

[11] NICHOLS, J.; FAULRING, A. Automatic interface generation and future user interface tools. *Workshop on The Future of User Interface Design Tools*, ACM, Pittsburg, 2005.

[12] SHIFFMAN, S.; DEGANI, A.; HEYMANN, M. Uiverify - a web-based tool for verification and automatic generation of user interfaces. In: *Proceedings of the 8th Annual Applied Ergonomics Conference. New Orleans, LA*. [S.l.: s.n.], 2005.