HAL
open science

Managing structure complexity in a multi-physic simulation software

Quoc Hung Huynh, Yves Maréchal, Jean-Louis Coulomb

To cite this version:

Quoc Hung Huynh, Yves Maréchal, Jean-Louis Coulomb. Managing structure complexity in a multi-physic simulation software. IEEE Transactions on Magnetics, 2006, 42 (4), pp.1239 à 1242. 10.1109/tmag.2006.870938 . hal-00189514

HAL Id: hal-00189514

https://hal.science/hal-00189514v1

Submitted on 25 Jun 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Managing Structure Complexity in a Multi-Physics Simulation Software

Huynh Quoc Hung, Yves Marechal, and Jean-Louis Coulomb

Laboratoire d'Electrotechnique de Grenoble, UMR 5529 INPG/UJF-CNRS, ENSIEG-BP 46-38402 Saint-Martin-d'Hères Cedex, France

This paper presents an efficient method for managing the complexity of software structure by implementing the business rules over the data model using a combination of logical programming and object-oriented programming, concretely applied in a multiphysics application.

***Index Terms*—Business rules, logical programming, multiphysics, object-oriented programming, simulation software.**

I. Introduction

THE development of numerical simulation software, particularly those involving electromagnetisms and other disciplines, implicitly increases the complexity of its structure. This increase demands a well-organized and reusable structure. In order to establish such a model, managing the complexity of software's structure should be observed.

Moreover, in the software community, especially in the numerical simulation community, it is well-known that the time spent to develop a whole application is much larger than that devoted to innovative aspects, such as the development of new algorithms with their related data. However, most of applications have a set of common functionalities, like database management, visualization, graphic and command user interface, etc. Hence, it is possible to increase software productivity by introducing reusability and extensibility.

The aim of this paper is to study the possibility of structure management by integrating a set of rules over the data model with help of the logical programming and the object-oriented programming.

II. Physics Description Structure

Our approach rises from a solution for the management of multiphysic solver structure [1]. The solver contains two parts: the description of physics and the description of resolution. We observe only the first part, which involves a database structure composed by many entities serving to describe the physics.

Fig. 1 shows a simplified global structure which contains the most important objects (*study, discipline, region, domainType, material, and property*) of our model. A study related to a discipline contains one or more regions, each of them relates to a domain type and has a material, which, at last, includes one or more properties. In fact, this model is common to all physics and should be extended, by inheritance mainly, to describe one or more physical disciplines.

Let us consider the example of the magneto-static case. The entities related to this physics will be introduced as subclasses. An advantage of this type of classification is that the subclasses can inherit all properties of their abstract classes. Fig. 2 presents the data model with added objects in our example.

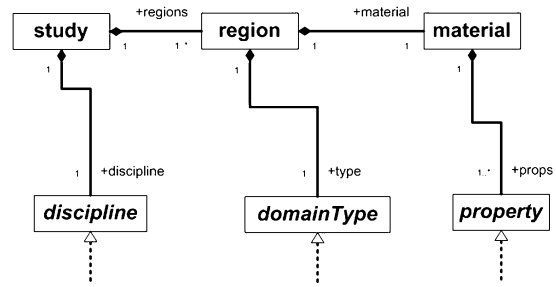

Fig. 1. Simplified global structure of the physics description.

III. From One to Several Physics

As a next step, consider now two physics: the magneto-static and the thermal with its new entities (shaded objects in the Fig. 3).

While the software development leads to an unavoidable increase of the structure's complexity, the global organization of the data model isn't modified. In fact, the number of realizations of these abstract structures may be important. For example, there are about 5000 terminal classes in Flux [10].

On the other hand, such an application needs a consistency checker to ensure that any command or object construction is valid [4]. For example, a magnetic region needs material with at least one magnetic property. This is not expressed by the data model that allows a mix of thermal regions with magnetic properties for instance. Thus, some constraints will be imposed over the data model. Let us consider for instance the magnetic aspects.

In order to implement these rules, the chosen language is naturally an international standard: object constraint language (OCL) [5] which has been developed to enhance the unified modeling language (UML) with constraints. Two examples of rules, associated to the Fig. 3 diagram in the magneto-static case, implemented in OCL will be presented. These rules ensure the validation of a choice when being in the magneto-static study.

1) The discipline of study must be of type *disciplineMagStat* and all regions must be of type *domainMagStat*:

Digital Object Identifier 10.1109/TMAG.2006.870938

0018-9464/$20.00 © 2006 IEEE

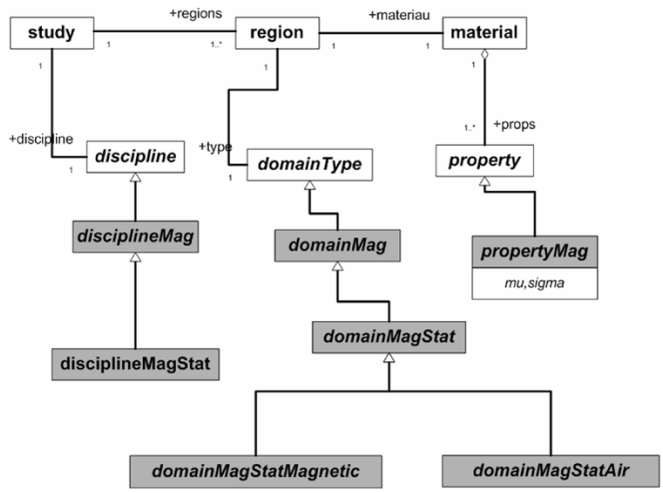

Fig. 2. Data model with added classes in magneto static case.

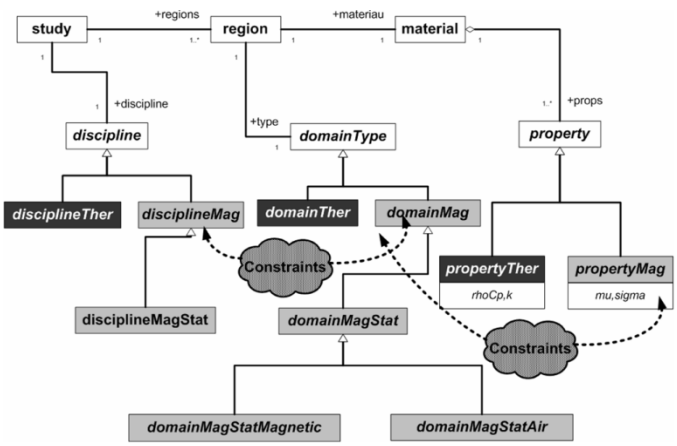

Fig. 3. Data structure in the two-physics case with constraints implementation.

context *study* ***def***:

let $caseDomain(d : OclType, t : OclType)$:

$\boldsymbol{Boolean} = (d = disciplineMagStat\ \boldsymbol{and}\ t = domainMagStat)$

let $validAllType(d : discipline, t : domainType)$:

$\boldsymbol{Boolean} = caseDomain(d \cdot type, t \cdot type)$

$\boldsymbol{or}\ t \cdot type \cdot supertypes$

$-> \boldsymbol{exists}(type : OclType | caseDomain(d \cdot type, type))$

context *study* ***inv*** *typeOfDomain*:

$\boldsymbol{self} \cdot \boldsymbol{regions} -> \boldsymbol{forAll}(r : region | validAllType(\boldsymbol{self} \cdot discipline, r \cdot type)$

2) The region must be of type *domainMagStat* and among the properties of the region's material; at least one is of type *propertyMag*.

context *region* ***def***:

let $caseProp(t : OclType, p : OclType)$:

$\boldsymbol{Boolean} = (t = domainMag\ \boldsymbol{and}\ t = propertyMag)$

let $validProp(t : domainType, p : property)$:

$\boldsymbol{Boolean} = caseProp(t \cdot type, p \cdot type)$

context *region* ***inv*** *propertyOfMaterial*:

$\boldsymbol{self} \cdot material \cdot props -> \boldsymbol{exists}(p : property \mid validProp(\boldsymbol{self} \cdot type \cdot type, p \cdot type))$

However, OCL is not a programming language so we need to use either coding or a logical programming language like PROLOG [6]–[9]. Indeed, to reduce the simulation software designer task, the consistency management should not require any hand coding. Moreover, PROLOG offers the possibility of inversibility. These are the main reasons explaining our translation of the rules formalism into a PROLOG clause.

IV. Business Rules in Object-Oriented Systems

Before passing to the rules implementation step, we should talk about the notions of business rules.

In an object-oriented model, business rules can be represented in many ways. Some rules will define the types of entities in a system along with their attributes. Other rules may define legal subtypes, which is usually done through subclassing. Other rules will define legal types of relationships between entities, which is applied in our case. These rules can also define basic constraints such as the cardinality of relationships and if a certain attribute is required or not [2], [3]. Most of these types of rules deal with the basic structure.

Our data model concerns in a developing structure and it becomes sophisticated gradually. In order to simplify the management of such a complex structure, the rules implementation will be an efficient solution.

V. Constraints Integrated in Data Model

First of all, two requirements on the implementation of the rules in the model will be presented. 1) The rules management must be as simple as possible and independent of the code. This aspect concerns the maintenance point of view. 2) The rules have to be not only evaluatable but also invertible. This will allow not only the validation of the rules for any input entered by the end-user but also the proposition of valid choices to the end-user if required.

These two requirements also explain the reasons why we have chosen to couple PROLOG, a logical and declarative programming language, to the object-oriented language.

In order to translate OCL expressions into PROLOG predicates, we have defined for each OCL syntactic element an equivalent PROLOG predicate. If we turn back to the two examples of the previous section regarding the magneto-static case, they are now expressed in PROLOG.

1) $caseDomain(disciplineMagStat, domainMagStat).$

$validAllType(Discipline, Domain)$:-

$isTypeOf(Discipline, DisciplineType),$

$isKindOf(Domain, DomainType),$

$caseDomaine(DisciplineType, DomainType).$

$inv(\boldsymbol{typeOfDomain}, \text{`}study', Study)$:-

$getField(Study, \text{`}discipline', Discipline),$

$getField(Study, \text{`}regions', Regions),$

$forall(Regions, R, (getField(R, \text{`}type', DomainType),$

$validAllType(Discipline, DomainType))).$

2) $caseProp(\text{`}domainMag', \text{`}propertyMag').$

$validExistProp(Domain, Prop)$:-

$isKindOf(Domain, DomainType),$

$isKindOf(Prop, PropType),$

```
        caseProp(DomainType, PropType).
inv(propertyOfMaterial, 'region', Region):-
        getField(Region, 'type', Type),
        getField(Region, 'material', Material),
        getField(Material, 'props', Props),
        exist(Props, P, (validExistProp(Type, P))).
```

From this point, we just need to add some lines of rules for each new physic. For example, in the case of thermal physics, just some rules lines should be added.

```
caseDomain('disciplineTher', 'domainTher').
caseProp('domainTher', 'propertyTher').
```

Otherwise, PROLOG, with its capacity of invertible queries, gives us the possibility of not only validation (initial aim of OCL rules) but also proposition and explanation for input data.

- Data input by users will be checked for the validation. For instance, in our magneto-static case, the discipline and the domain type of a given study are verified by applying the rules to assure their compatibility.
- When a field is not filled in, the potential field values will be proposed to complete the object creation. For example, if we haven't chosen a domain type for a region, two potential domain types will be proposed. In our magneto-static case, they are *domainMagStatMagnetic* and *domainMagStatAir*
- We have added a predicate "why" in order that the errors would be explained. We consider two examples.

1) If the discipline of a study is not given, the error will be notified by a message

```
why(typeOfDomain, 'study', Study, Why):-
        getField(Study, 'discipline', null),
        Why = ['Field discipline of study is not filled', '\n'].
```

2) If the discipline of a study is not given, the error will be notified by a message

```
why(typeOfDomain, 'study', Study, Why):-
        getField(Study, 'discipline', Discipline),
        Discipline\ == null,
        reference(Discipline, DisciplineRef),
        Why = ['Study discipline and domain type are not
        compatible', '\n'].
```

Considering a simple example described in the tables below, we have three regions with their domain type and material. In the second table, we have three studies with their regions and their discipline defined by user. We will try to verify these studies and let the engine propose potential values for incomplete or incorrect fields in Table I.

- Verifying the validation of Study MagStat

```
?- check('study', 'Study MagStat', Obj).
→ Study MagStat is valid.
```

TABLE I
POTENTIAL VALUES FOR INCOMPLETE OR INCORRECT FIELDS

region	domainType	material
air	*domainMagStatAir*	*vacuum*
circuit	*domainMagStatMagnetic*	*steel*
core	*domainTher*	*steel*

study	regions	discipline
Study MagStat	*air,circuit*	*Discipline MagnetoStatic*
Study Ther	*core*	*X*
Study Ther_1	*core*	*Discipline MagnetoStatic*

```
→ Obj = study('Study MagStat', [regions =
[region(air), region(circuit)],
        discipline = disciplineMagStat('Discipline MagnetoStatic')])
```

- Verifying the validation of Study Ther

```
?- check('study', 'Study Ther', Obj).
→ Field discipline of study is not filled.
→ Obj = study('Study Ther', [regions = [region(core)], discipline =
null])
```

- Proposing the incomplete field "discipline" of Study Ther

```
?- propose('study', 'Study Ther',
        study('Study Ther', [regions =
[region(core)], discipline = Discipline])).
→ Discipline = disciplineTher('Discipline Thermic')
```

- Verifying the validation of Study MagStat_1

```
?- check('study', 'Study Ther_1', Obj).
→ Study discipline and domain type are not compatible.
→ Obj = study('Study Ther', [regions = [region(core)],
        discipline = disciplineMagStat('Discipline MagnetoStatic')])
```

- Proposing the incorrect field "discipline" of Study MagStat_1

```
?- propose('study', 'Study Ther_1', study('Study Ther_1',
        [regions = [region(core)], discipline = Discipline])).
→ Discipline = disciplineTher('Discipline Thermic').
```

Once completely constituted, the rules need to be implemented in the software. Fig. 4 presents the process of database treatment with rules implementation. The entry data through data input user interface is stored temporarily before the checker verifies data validation. The validation checker is described with more details in the Fig. 5.

Thanks to a *database of OCL operators written in PROLOG*, a rules interpreter establishes the communication between *graphical user interface (GUI)* and the *business rules in OCL*. Inside the rules interpreter, a *parser* helps us to transform PROLOG rules into OCL business rules. These rules will be applied over the data model.

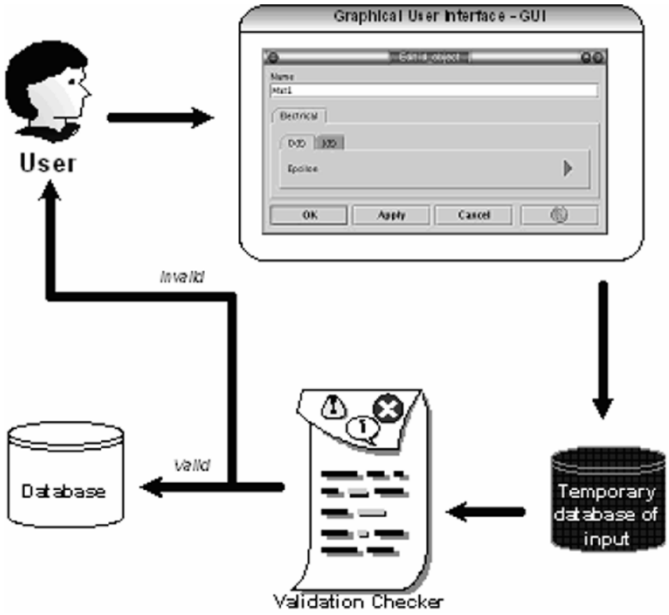

Fig. 4. Database treatment process with the implementation of rules.

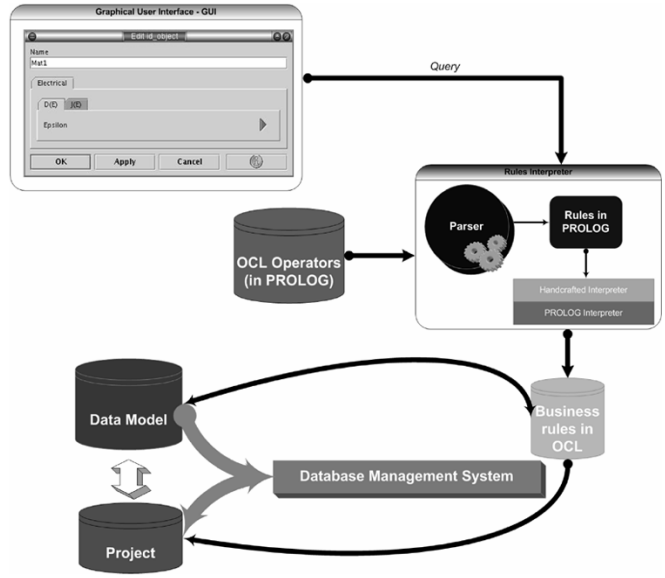

Fig. 5. Rules integration in database structure—validation checker description.

VI. Application and Perspectives

We have implemented the rules created to support the description of our simulation software FLUX. The constraints integrated will verify the validation of objects construction. It is believed that this approach will give a possibility to manage the structures which contain complex entity relationships.

This approach can be developed by reaching a dynamic business rules implementation with adaptive object-models, as proposed in [2] and [3].

VII. Conclusion

We have proposed a solution for managing the complexity of the data model in a multiphysics solver by implementing a set of rules into the data model of a physics descriptor in our multiphysics solver. We have translated the unambiguous syntactic belonging to the standard object-oriented UML and OCL into PROLOG, a logical programming language. Hence, the proof-mechanism is reusable for any model described with this standard.

The goal of this proposition is to permit developers to efficiently manage, modify and execute the software data model. The use of PROLOG allows us to satisfy the requirements on the rules implementation in our model: the simplicity and the code independence. As a result, the time spent to describe a physical problem is reduced. Besides, the rules permit to establish a well-organized data structure, which naturally simplifies the work of a physicist who wants to add a new discipline in the global structure without modification.

References

[1] H. T. Luong and Y. Maréchal, "Modélisation numérique multiphysique et multiméthode: une description formelle pour construire un environnement de simulation général," *Proc. MajecSTIC 2003*, p. 23, Oct. 2003.

[2] J. W. Yoder and R. Johnson, "Metadata and adaptive object-models," in *ECOOP'2000 Workshop Reader—Lecture Notes in ComputerScience*. New York, 2000.

[3] ——, *Implementing Business Rules with Adaptive Object-Models*. Englewood Cliffs, NJ: Prentice-Hall, 2002.

[4] O. Defour and Y. Maréchal, "Static and dynamic consistency checking for numerical simulation: A mixed logic and object oriented programming approach," *IEEE Trans. Magn.*, vol. 40, no. 2, pp. 1642–1645, Mar. 2003.

[5] OMG documents—"Unified Modeling Language, v1.5—Object Constraint Language Specification" (2003, Mar.). [Online]. Available: http://www.omg.org

[6] M. A. Covington, "Some Coding Guidelines for PROLOG," Artificial Intelligence Center, Univ. Georgia, Atlanta, Dec. 2, 2002.

[7] J. R. Fisher. (1999–2004) PROLOG Tutorial. California State Polytechnic Univ., Pomona, California. [Online]. Available: http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/

[8] M. Banbara and N. Tamura. (2003, Feb.) PROLOG Cafe Documentation version 0.6. [Online]. Available: http://kaminari.istc.kobe-u.ac.jp/PROLOGCafe/PROLOGCafe061/doc/

[9] J. Wielemaker, SWI-PROLOG 5.4.1 Reference Manual, Dept. Social Science Informatics (SWI) Roeterstraat, 2004.

[10] *Flux Documentation—"Notice d'utilization générale de Flux 3D, version 8.1"—CEDRAT*, 2003. Juillet.

Manuscript received June 28, 2005 (e-mail: Hung.Huynh-Quoc@leg.ensieg.inpg.fr).