Model Driven Ecological Interface Creation: The Constraints Model

Alexandre Moïse, Jean-Marc Robert

Département de mathématiques et de génie industriel
École Polytechnique de Montréal
C.P. succ. Centre-Ville, Montréal (Québec)
H2C 3A7 Canada
{alexandre.moise, jean-marc.robert}@polymtl.ca

Abstract. An ecological interface (EI) consists in a user interface (UI) displaying the constraints of a specific environment in order to support a user's decision-making process. The design gap problem can be decomposed in three broad categories: incomplete process description, confusing and lacking representations, lack of integration with a software internal structure. In order to address these problems, a model driven engineering approach is applied to the EI creation process, resulting in a model driven EI creation method. One component of this method is the constraints model. This paper presents both the abstract and concrete syntaxes of its modeling language. The abstract syntax enables a more formal definition of EI concepts for which the EI literature does not offer an operational definition making their translation into software components difficult. While many application of domain specific modeling (DSM) emphasize an increase in productivity, this application shows that DSM efforts enable a better understanding of an ill-defined domain.

Keywords: Model driven engineering, domain-specific language, user interface, ecological interface.

1. Introduction

An ecological interface (EI) consists in a user interface (UI) displaying the constraints of a specific environment in order to support a user's decision-making process. Contrarily to a traditional UI which, to represent the state of a particular environment, emphasizes on displaying pieces of information, an EI rather emphasizes on the relations between these pieces of information while displaying them at the same time. Results of experiments comparing prototypes of EIs with prototypes of traditional UIs supporting fault detection and diagnosis tasks in various application domains show the superiority of the former over the latter [23].

1.1 The Design Gap

The problem this paper partially addresses is what is referred to as the design gap. The EI design gap problem can be described alongside three dimensions: process, representations, and integration.

Process. The EI literature provides barely any explanation on how to go from an analysis model to the implementation of an EI. According to the EI literature, every EI creation[1] process begins with an analysis model representing the constraints of the environment. This model takes the form of an abstraction-decomposition hierarchy (ADH) [1], [22], [24]. From this point, [1] state that elements of the ADH can be

[1] Although the term usually employed is *ecological interface design*, it seems more appropriate to use the term *ecological interface creation*. As strange as it can appear, every approach presented in the literature based on this theoretical model (e.g., [1] and [22]), have put more emphasis on the analysis phase rather than on the design phase [25]. Since this is about a complete EI creation process—from requirements to implementation—the term *design* seems too limited and is therefore replaced by the term *creation* for the rest of this paper.

mapped to visual components, each representing a particular constraint of the environment. However, [1] do not define this mapping nor the steps required in a very clear way, thus requiring creativity and judgment from designers. Furthermore, there is no indication on how these visual components are translated into software components.

Representations. A representation of the constraints of the environment is the root artifact of the EI creation process [1], [22], [24]. To this end, the ADH has been the only form of representation put forth in the EI literature [1], [14], [22], [24]. However, the same literature presents many problems related to the ADH apparently inhibiting its widespread use among practitioners which include the following:

- The creation of an ADH for a particular environment is costly in time and effort [23].
- Passing from an ADH to an EI demands a fair deal of creativity—it could be considered more of an art than a science [23].
- It is far from being easy to gain a fair understanding of the semantics of an ADH for a non-expert [7]. Validation is therefore very difficult—sometimes even impossible—with experts of the application domain to which the environment belongs.
- The ADH needs to be detailed by other forms of representation of the environment [1], [7]. For example, [1] suggest creating a list of variables and a list of equations.

Aside from the ADH, which consists of representing elements related to the analysis phase, no other form of representation has been presented in the EI literature. Thus, design decisions related to the EI appearance are not represented in a standard way creating potential communication difficulties between designers and implementers.

Integration. A software can be decomposed in three types of components: UI, logic, and data. Data can be defined as a representation of information used in logic processing. Logic can be defined as the transformation of data into other data. A UI allows a user to trigger certain types of logic processing and display its result. A UI is thus a part of a software and cannot be considered without the other parts. Nonetheless, the EI literature does not present any software component architecture or framework. Therefore, there is no indication on how to integrate an EI with the internal structure of a software.

1.2 Model Driven Engineering

Model driven engineering (MDE) consists mainly in defining a successive set of models as well as transformation rules allowing to obtain a target model from a source model[2]. This approach to software development implies specification of modeling dimensions, mappings between models, a process, and a set of software tools to support modeling and automated transformations between models [5]. Ultimately, an MDE approach seeks to increase productivity. This objective results from the combination of the following advantages[3]:

- *Managing complexity*. Mappings between models from within one dimension or from across different dimensions provide traceability between models. It is thus easier to understand the impact of a change in a model on other models.
- *Artifact reuse*. Models can be reused in different projects, thus do not have to be recreated. For example, the same domain model can be reused for different technological platforms.
- *Improved artifact quality*. Transformations from one model to another are based on a set of formal rules. As long as these rules stay unchanged, target artifacts will always be created the same way.

In light of these advantages, it is the authors' position that the design gap can be shortened by applying a MDE approach to the EI creation process, resulting in a model driven EI creation method. The question is therefore how would such a method look like and how would it fit with the creation of other software components? Figure 1 presents an overview of a possible set of models for a three-tier application based on Model Driven Architecture (MDA) [2], probably the best known MDE approach. Since an EI is a particular type of UI, it is related to the presentation tier at the Platform Specific Model (PSM) level. At the Platform

[2] Source code is considered to be a model since it is a representation of the software execution.
[3] These advantages are found directly or indirectly in [2], [5], [6], [8], [16], [17], and [19].

Independent Model (PIM) level, the domain model is used to represent the application domain rules, i.e., how are elements of the application domain conceptually related to each other. For instance, especially with a MDA approach, UML class diagrams and Object Constraints Language can be used to represent the domain model. Using transformation patterns, the domain model can be transformed into different PSM-level models such as the presentation model which can be divided into the abstract presentation model and the concrete presentation model [21]. The former consists mainly in representing the information structure of the UI, interaction mechanisms, and relations with other software component. The latter consists in a representation of the UI as it will be displayed to the user. As such, an EI creation method would define an abstract presentation model and a concrete presentation model as well as transformation rules between the domain model and the abstract presentation model, between the abstract presentation model and the concrete presentation model, and between the concrete presentation model and the presentation source code. A comprehensive description of this method would take much more then the allocated maximum length for this paper, consequently only a component of the method is presented: the abstract presentation model, called the *constraints model*.

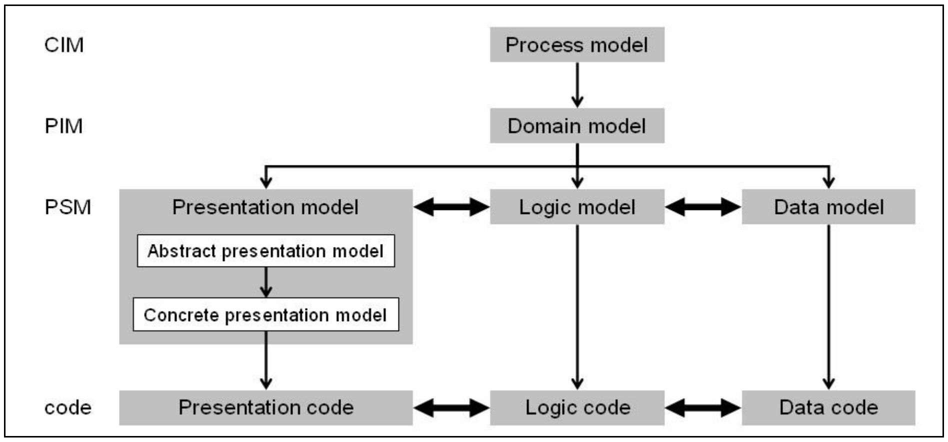


Fig. 1. Overview of a possible set of models for a three-tier application based on Model Driven Architecture.

The constraints model consists in a representation of the constraints of the environment, which, as stated previously, is the starting point of any EI creation process. Since the ADH exhibits many problems, this model is proposed as a replacement. The constraints model must therefore exhibit the same features as the ADH.

1.3 Outline

Section 2 gives a brief description of what is an EI. The focus is on describing the features of the ADH in order to validate that the constraints model exhibits them. Section 3 describes the proposed modeling language for the constraints model. Both the abstract syntax, i.e., its meta-model, and concrete syntax are described. Section 4 presents an example of the application of the modeling language. The validation of the features of the constraint model is achieved through this example.

2. What is an Ecological Interface?

2.1 The SRK Taxonomy

According to [24], EI design is a theoretical framework based on the SRK taxonomy [14]. This taxonomy consists in a qualitative model representing three interconnected levels of cognitive control. Each level is based on a particular type of internal representation of the environment: skill-based behavior (SBB), rulebased behavior (RBB), and knowledge-based behavior (KBB). The recourse to one level rather than another is determined by the way a person interprets perceived information.

The purpose of an EI is twofold. On the one hand, it should allow its user to complete a task at the lowest possible level of cognitive control. On the other hand, it should support all three levels of cognitive control. In order to do so, [24] present three design principles, each related to one of the three levels of cognitive control:

- **SBB.** In order to correct a situation, the user should be able to manipulate objects of the environment directly on the UI. This calls for the direct manipulation interface paradigm [18].
- **RBB.** In order to support fault detection tasks, each constraint of the environment must be explicitly displayed by the UI. A constraint is a relation between different pieces of information allowing to assess if the representation of the environment displayed by the UI is or is not in a normal state. If a constraint is not respected, the environment is said to be in an abnormal state.
- **KBB.** In order to support diagnosis tasks, the UI must represent the functional hierarchy of the environment. In order to do so, relations between every piece of information used to describe the state of the environment must be explicitly presented by the UI.

2.2 Representation of the Environment

As stated previously, a representation of the constraints of the environment is at the foundation of any EI creation process [1], [22]. The constraints rely on a structure of the environment, composed of the many concepts related to a particular environment as well as the relations between them. According to [10], this structure imposes constraints on the set of possible actions intended to achieve a predetermined goal—it will be possible to achieve this goal only if the actions respect the constraints of the environment. In the case of an EI, [9] proposes that a constraint be defined as an equation or an inequality, i.e., a relation whose nature is represented by a logical operator.

2.3 The Abstraction-Decomposition Hierarchy [4]

In any software development process, the analysis phase aims at describing the application domain. It is a process in which a perception of the real world is transformed into a representation [12]. This representation, in the form of a model, is generally conceptual in nature, i.e., without any consideration of the technologies that will be used to implement the software. In an EI creation process, the analysis phase consists of modeling the constraints of the environment and its underlying structure.

As stated previously, the ADH is the only form of representation of the constraints of the environment put forth in the EI creation literature [1], [14], [22], [24]. The ADH is not concerned with tasks nor tools, but with goals and information (variables and relations) required to the understanding of the dynamics of the environment. It consists in structuring all pieces of information essential to achieving goals through different levels of abstraction and decomposition. According to [24], the ADH is psychologically compatible with a person’s problem-solving process. Figure 1 illustrates the generic frame of an ADH.

Each level of abstraction describes a set of concepts—for which each one can be described by a collection of interrelated variables—and relations corresponding to that level. As represented in figure 1, the links between levels of abstraction are “means-ends” relations, which means that for any vertical pair of levels of abstraction, the concept of the higher level describes the end to achieve (the “why?”) and the associated concepts of the lower level describe the means (the “how?”) to achieve this end. The highest level of abstraction describes the goals perused by the manipulation of the environment and the lowest level of abstraction generally describes physical aspects of the environment, i.e., those that can be manipulated or are perceivable. Intermediary levels of abstraction relate to functional aspects of the environment between these two levels of abstraction. In other words, going from one level of abstraction to another involves a change in concepts and representation structure as well as a change in variables used to describe the environment at this level of abstraction [14]. This set of relations between levels of abstraction is called a

[4] For a comprehensive definition of the ADH, see [1], [14], [15], and [22].

functional hierarchy. Usually, the ADH is divided in five levels of abstraction, but this number can vary depending on the application domain [24].

As for the decomposition dimension, it allows to divide the representation of the environment under analysis into sub-sets of components that are responsible for its good functioning. It consists in decomposing the parts from their whole. Thus, a concept represented in a particular column is composed of the concepts in the column immediately to its right. Inversely, a concept represented in a particular column is a part of a concept represented in the column immediately to its left. For example, in a medical context, the representation of a patient can be divided in the following way, in the order of columns from left to right: body, system, organ, tissue, and cell [3].

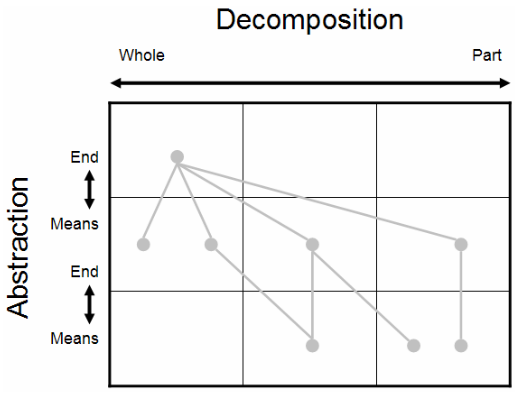


Fig. 2. Generic frame of the abstraction-decomposition hierarchy.

Information obtained from this analysis is concerned with supporting RBB and KBB levels of cognitive control. In order to support KBB, the UI must display the functional hierarchy of the environment to the user. The functional hierarchy contains all the concepts and their underlying variables being used to describe the environment as well as the relations between these variables. With regard to supporting RBB, it is necessary that the UI displays the constraints of the environment, i.e., the relations between the underlying variables of every concept, in order to quickly detect a fault in the state of the environment. As for how to support SBB, it must be specified at a later phase of the EI creation process, more precisely, when completing the concrete presentation model, i.e., when decisions on visual aspects and interactions will be specified.

3. The Constraints Model [5]

The literature in UI engineering research shows interest in modeling different aspects related to a UI such as context of use [19], task [13], interactions [11], as well as navigation and information organization [4]. These modeling languages have been created only for UI aimed at supporting tasks with a predefined sequence of actions, e.g., hotel room reservation or checking out a book from a library. The constraints model presented in this section is intended for EI creation, i.e., a UI aimed at supporting its user's decision-making process, more precisely, fault detection and diagnosis tasks.

The following two subsections describe respectively the abstract syntax and the concrete syntax of the constraints model. The abstract syntax of a modeling language consists in a model of its elements and how they are related to each other. These models are usually called meta-models since they are models explaining models. As for the concrete syntax, it consists in a set of symbols representing the elements of the abstract syntax. These symbols are used to create models.

[5] [9] propose a modeling language for the abstract presentation of an ecological interface. The abstract syntax described in this section constitutes an improved version of the one presented in [9], while the concrete syntax is essentially the same.

3.1 Abstract Syntax

In the first subsection, a meta-model, represented with UML class diagrams, is used to describe the abstract syntax. In order to have a clearer view of the meta-model, it is decomposed in various packages. The second subsection presents possible extension mechanisms of this meta-model.

The modeling language used to represent the meta-model has been chosen based on two reasons. The first is that, relying on the quantity of publications on the subject, a UML class diagram seems to be a well-known and well-understood notation. The second is that tools implementing this meta-model will be developed using object-oriented (OO) technologies and there is a direct mapping between OO components and UML class diagrams. Aside from these reasons, the meta-model could have been represented using any modeling language. What is of essence is that elements of the meta-model and their relations to each other can be represented in an intelligible manner.

The Meta-Model. As stated previously, the distinguishing feature of an EI, compared to a traditional UI, is its ability to display relations between pieces of information that can be referred to as variables. If a relation between variables can take the form of an operation, the nature of this relation can therefore be represented by an operator. The main element of an EI is thus the *operator*. It is by making operations visually explicit that a UI can support a user's RBB and KBB levels of cognitive control. An operation cannot be represented without the *variables* used in its computing. Variables are usually related to a *concept*.

Figure 3 illustrates the base meta-model. Every instance of these elements has a unique code of identification. The relation between concepts as well as between a concept and a variable is shown in this figure. Although there exists a relation between a variable and an operator, this relation is manifested differently depending on the types of the variable and the operator. This relation is shown in subsequent figures illustrating different packages of the meta-model.

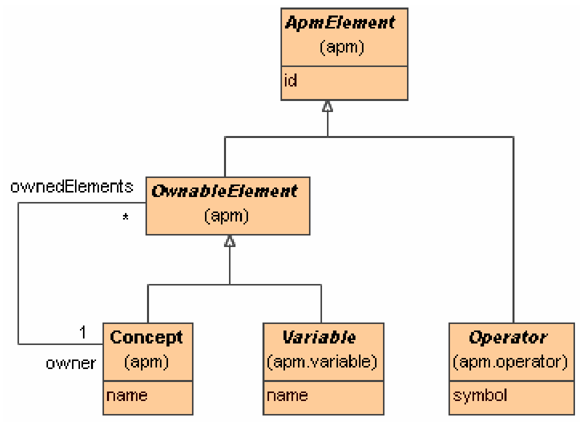


Fig. 3. Base meta-model from package `apm` (abstract presentation model).

Figure 4 illustrates the concrete variable types. A variable has a name and can be manipulable or not. If the value of the `isManipulable` attribute is set to `true`, the value of this variable is set by the user. If, on the other hand, the value of this attribute is set to `false`, the value of this variable is set by the software.

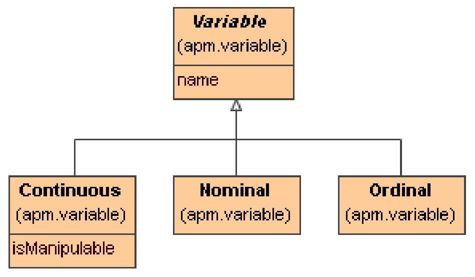


Fig. 4. Variables meta-model from package `apm.variable`.

Figures 5, 6, 7, and 8 illustrate the different types of operator. A symbol is assigned to every type of operator. An operator can be a logical operator, an arithmetic operator or an aggregation operator. Each type is subsequently described.

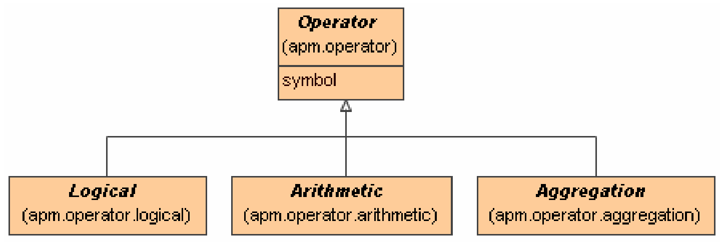


Fig. 5. Operators meta-model from package `apm.operator`.

An aggregation operator is one that takes into account a collection of values of its unique operand of variable type `Continuous`. Its computation returns a unique numerical value.

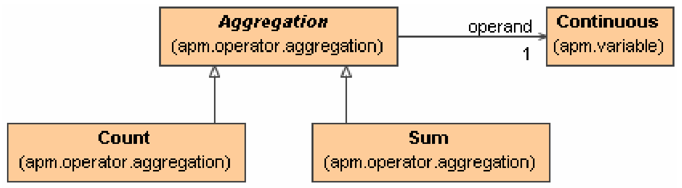

Aggregation
(apm.operator.aggregation)
operand
1
Continuous
(apm.variable)
Count
(apm.operator.aggregation)
Sum
(apm.operator.aggregation)

Fig. 6. Aggregation operators meta-model from package `apm.operator.aggregation`.

An arithmetic operator is one that computes a value out of two values, called operands. The operands are instances of a `Continuous` type of variable. The order of the two operands is important for certain types of arithmetic operators such as `Subtraction` and `Division`.

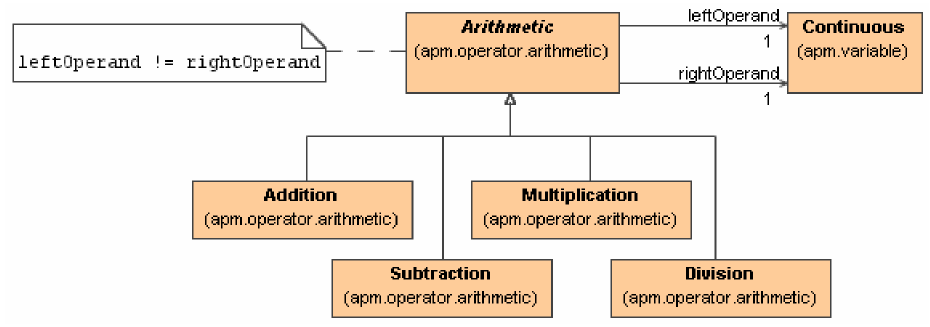

leftOperand != rightOperand
Arithmetic
(apm.operator.arithmetic)
leftOperand
1
rightOperand
1
Continuous
(apm.variable)
Addition
(apm.operator.arithmetic)
Multiplication
(apm.operator.arithmetic)
Subtraction
(apm.operator.arithmetic)
Division
(apm.operator.arithmetic)

Fig. 7. Arithmetic operators meta-model from package `apm.operator.arithmetic`.

A logical operator is one that returns a Boolean value, i.e., either true or false when compared to a result, an instance of a `Continuous` type of variable. The right side can take many forms which are subsequently described in figure 9.

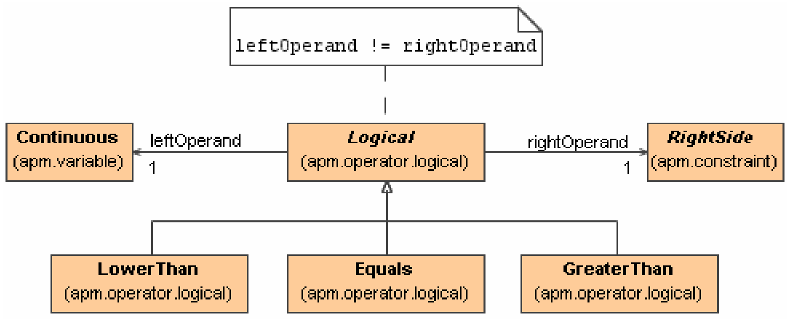


Fig. 8. Logical operators meta-model from package `apm.operator.logical`.

Finally, figure 9 illustrates the constraint meta-model. A constraint is an operation that represents either an equation or an inequality. It is composed of a logical operator whose left operand is a continuous variable and whose right operand can be a continuous variable, the result of an aggregation operator, or the result of an arithmetic operator. A constraint would be said to be broken if the result of the logical operator would yield false. Every constraint is assigned to a level. The collection of constraints ordered by level constitutes the functional hierarchy.

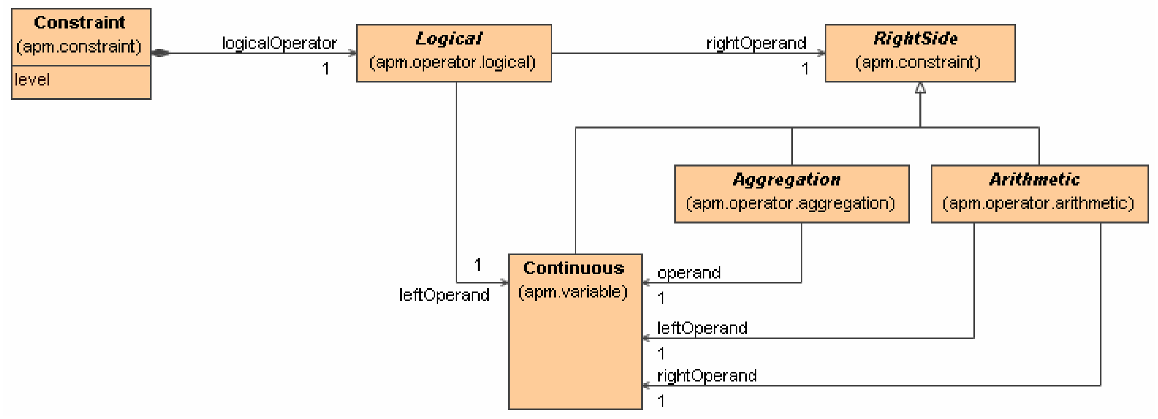


Fig. 9. Constraint meta-model from package `apm.constraint`.

Meta-Model Extension. It is possible to extend the meta-model by adding meta-classes, abstract or concrete, to its inheritance hierarchy. For example, the aggregation meta-model (figure 6) can be extended by adding new concrete meta-classes such as `Average` or `Maximum` and the logical meta-model (figure 8) can be extended by adding new concrete meta-classes such as `EqualOrGreaterThen` or `EqualOrLowerThan`.

3.2 Concrete Syntax

The previous section has described the abstract syntax of the constraints model. This section aims at describing the concrete syntax in order to be able to visualize all three basic model elements: concept, variable and operator.

Figure 10 illustrates the concrete syntax of a concept. The gray upper rectangle contains the name of the concept. The white lower rectangle contains the owned elements and can be resized.

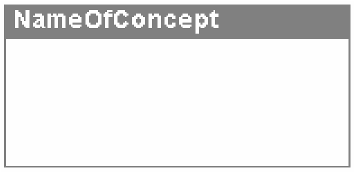


Fig. 10. Concrete syntax of a concept.

Figure 11 illustrates the concrete syntax of a variable. The left rectangle contains the variable type. If it is an instance of the `Ordinal` meta-class, a "O" is displayed, if it is an instance of the `Continuous` meta-class, a "C" is displayed, and if it is an instance of the `Nominal` meta-class, a "N" is displayed. The right rectangle contains the name of the variable. If the name section has a solid border, it means that the variable is not manipulable by the user. On the other hand, if it has a dashed border, it means that the user can modify the value of this variable through the UI.

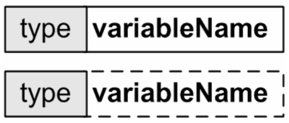


Fig. 11. Concrete syntax of a variable.

Figure 12 illustrates the concrete syntax of an operator. It is a circle with the symbol of the operator. For example, the letter "S" is replaced by "+" for an instance of the meta-class `Addition` or a "<" for an instance of the meta-class `LowerThan`. Instances of the meta-class `Operator` can be associated to instances of the meta-class `Variable`. These associations must nonetheless observe certain rules (figures 6, 7, and 8). For example, an instance of the `Arithmetic` meta-class must be associated to two instances of the `Continuous` meta-class, while an instance of the `Aggregation` meta-class must be associated to only one instances of the `Continuous` meta-class.

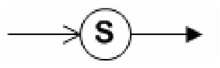


Fig. 12. Concrete syntax of an operator.

Since some operators require more than one variable and their order is important, there are two types of association, each defined by a different arrowhead. The "from" association is illustrated by the arrowhead at the left of the operator symbol in figure 12 and the "to" association is illustrated at its right. No matter where the variable and operator symbols are displayed, the arrowheads determine the direction in which the operation should be read. The "from" association is associated to the first (left) operand and the "to" association is associated to the second (right) operand.

4. Application of the Modeling Language

Now that the abstract syntax and the concrete syntax have been described, this section presents an example of the application of this modeling language, in other words, an example of a constraints model. Validating that the constraints model exhibits the features of the ADH will be achieved through this example.

The application consists in managing a mutual fund portfolio for a client. The portfolio manager needs to achieve two goals: (1) making sure that the client's target rate of return is lower then the rate of return of the portfolio and (2) making sure that the client's risk tolerance is greater than the risk of the portfolio. The constraints model for the portfolio management application is illustrated in figure 13.

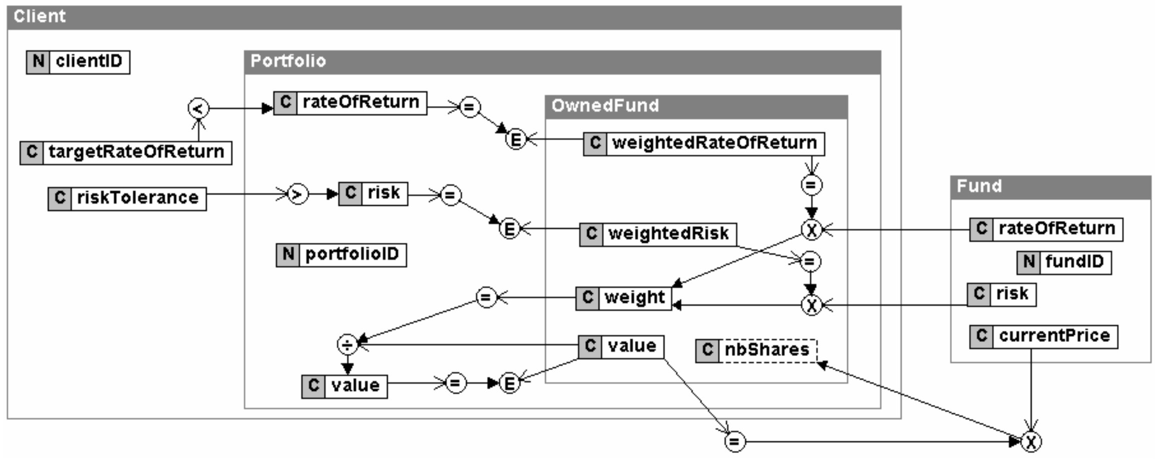
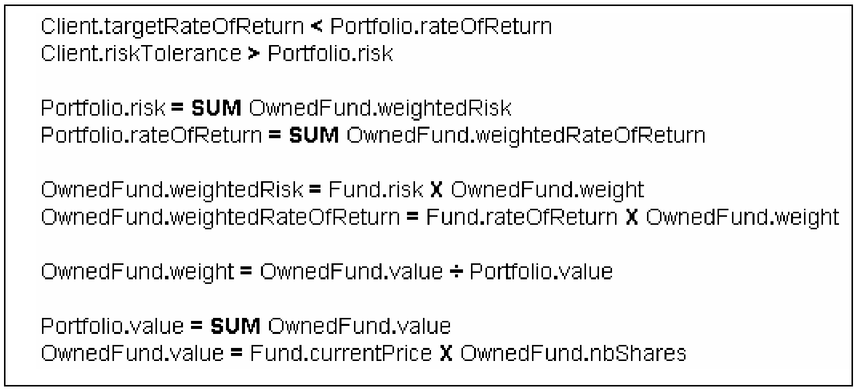


Fig. 13. Constraints model for the portfolio management application.

The constraints model presents, on the one hand, the structure of the environment, composed of concepts, variables, and how they are related to each other. These relations can take two forms: inclusions and operations. Inclusions indicate ownership between a concept (the owner) and either another concept or a variable (the owned element). For example, a client "owns" one or many portfolios, and a portfolio "owns" one or many funds (`ownedFund`). These visual inclusions are equivalent to the decomposition dimension of the ADH. As for operations, they link variables in a particular way.

On the other hand, the constraints model presents, as its name implies, the constraints of the environment. Each constraint, as defined by the constraints meta-model illustrated in figure 9, is composed of a logical operator and another type of operator. Constraints are based upon the structure of the environment.

The constraints set illustrated in figure 14 is generated from the constraints model illustrated in figure 13 using a two-part algorithm. The first part consists in searching through the operator collection and creating a constraint with every logical operator found. A constraint is well-formed if it has a logical operator with a left-side continuous variable and a complete right side as defined in figure 9. The second part of the algorithm consists in determining the constraint level. Considering that a constraint at the first and highest level is one whose left-side continuous variable is not found in the right side of any other constraint, a level is obtained by counting the depth of a left-side continuous variable. For instance, the constraint "`OwnedFund.weightedRisk` **=** `Fund.risk` **X** `OwnedFund.weight`" is of level three since `OwnedFund.weightedRisk`, the left-side operand, is part of the right side of constraint "`Portfolio.risk` **=** **SUM** `OwnedFund.weightedRisk`". The left-side operand of this constraint, `Portfolio.risk`, is part of the right side of constraint "`Client.riskTolerance` **<** `Portfolio.risk`", which left-side operand is not part of any constraint right side.



```
Client.targetRateOfReturn < Portfolio.rateOfReturn
Client.riskTolerance > Portfolio.risk

Portfolio.risk = SUM OwnedFund.weightedRisk
Portfolio.rateOfReturn = SUM OwnedFund.weightedRateOfReturn

OwnedFund.weightedRisk = Fund.risk X OwnedFund.weight
OwnedFund.weightedRateOfReturn = Fund.rateOfReturn X OwnedFund.weight

OwnedFund.weight = OwnedFund.value ÷ Portfolio.value

Portfolio.value = SUM OwnedFund.value
OwnedFund.value = Fund.currentPrice X OwnedFund.nbShares
```

Fig. 14. Constraints set obtained from the constraints model of the portfolio management application.

Constraints are grouped by level. These levels are equivalent to the abstraction dimension of the ADH in that each level explains the variables from the above level. Relations between variables of different levels thus constitute a functional hierarchy. The highest level represents the goals of the system. The two inequalities represent the two objectives a portfolio manager must achieve. They would be associated with the highest level of abstraction of the ADH illustrated in figure 2.

According to [24], a fault in an environment occurs when a constraint is broken. In other words, a fault occurs when an equation or an inequality yields false. If this is the case, the environment is in an abnormal state. In order to resolve the situation, the first step is to understand the cause of this abnormality, i.e., diagnose the problem. In order to do so, one has to navigate the functional hierarchy. The problem will be solved after modifying the value of the relevant manipulable variables that are usually associated to the lowest level of abstraction. For example, if the risk of the portfolio goes below the clients' risk tolerance, the portfolio manager must modify the number of shares (the manipulable variable) of owned funds since there is an indirect relation between these two variables. Since there is also an indirect relation between the number of shares and the portfolio rate of return, the portfolio manager must be aware of the impact of the modification on this goal-level variable.

5. Conclusion

An MDE approach seems a promising path to shorten the design gap problem of EI creation. To this end, this paper has presented both the abstract and concrete syntaxes of the constraints model, one of the components of a model driven EI creation method. Although the EI literature presents the ADH as the only form of representation of the constraints of the environment, the same literature presents a number of problems with it. To this end, the constraints model seems to be an acceptable replacement of the ADH since it apparently exhibits the same features while offering better mapping to implementation technologies, precisely OO programming languages concepts.

Many concepts related to EI creation are defined only at a conceptual level in the literature, namely "functional hierarchy" and "constraint". Describing the abstract syntax of the constraints model contributed not only to a better understanding of these concepts but mostly to an operational comprehension of them from a MDE perspective.

Future research related to a model driven EI creation method will consist in creating many of its components presented in figure 1. Firstly, as stated previously, a UI is a part of a software. Many commercial and open source tools allow the generation of source code of a complete software from a domain model [20]. It will therefore be necessary to define mappings between domain model elements and constraints model (abstract presentation model) elements in order to permit automated transformation from the former to the latter. Secondly, transformation from the constraints model to visual components based on the ones defined by [1] will have to be defined. These visual components are part of the concrete presentation model in which decision related supporting SBB level of cognitive control will be taken. Thirdly, transformation from the concrete presentation model to source code will have to be defined. This implies the creation of a component library. While some components will relate to the UI specifically, others will be responsible for connecting with the software internal structure.

References

1. Burns, C. M., Hajdukiewicz, J. R. *Ecological Interface Design*, CRC Press, 2004.
2. Frankel, D.S. *Model driven architecture*, Wiley Publishing, Inc, 2003.
3. Hajdukiewicz, J. R., Vicente, K. J., Doyle, D. J., Milgram, P., Burns, C.M. "Modeling a medical environment: an ontology for integrated medical informatics design", *International Journal of Medical Informatics*, Vol. 62, pp. 79-99, 2001.
4. Hennicker, R., Koch, N. "Modeling the user interface of web applications with UML", *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, Workshop of the pUML-Group at the UML 2001* (October 1-5, 2001, Toronto). 2001, pp. 158-173.
5. Kent, S. "Model driven engineering", *Proceedings of the Third International Conference on Integrated Formal Methods IFM 2002* (May 15-18, 2002, Turku, Finland), Springer, Berlin, 2002, pp. 286-298.
6. Kulkarni, V., Reddy, S. "Separation of concerns in model-driven development", *IEEE Software*, Vol. 20, No. 5, 2003, pp. 64-69.
7. Lind, M. "Making sense of the abstraction hierarchy", *Proceedings of the Conference on Cogntive Science Approaches to Process Control (CSAPC'99)*, 1999, pp. 195-200.
8. Mellor, S.J., Clark, A.N., Futagami, T. "Model-driven development", *IEEE Software*, Vol. 20, No. 5, 2003, pp. 14-18.
9. Moïse, A. , Robert, J.-M. "Un langage de modélisation pour la présentation abstraite d'une interface écologique," *7ième Congres International de Génie Industriel (CIGI2007)* (June 5-8, 2007, Trois-Rivières, Canada), 2007.
10. Newell, A., Simon, H. A., *Human Problem Solving*, Prentice-Hall, 1972.
11. Nunes, N.J., Cuhna, J.F. "Wisdom: a software engineering method for small software development companies", *IEEE Software*, Vol. 17, No. 5, 2000, pp. 113-119.
12. Odell, J., Ramackers, G. J. "Toward a formalization of OO analysis", *Journal of Object-Oriented Programming*, Vol. 10, pp. 64-68, 1997.
13. Pinheiro da Silva, P., Patton, N.W. "User interface modeling in UMLi", *IEEE software*, Vol. 20, No. 4, 2003, pp. 62-69.
14. Rasmussen, J. "Skills, Rules, and Knowledge; Signals, Signs, and Symbols, and Other Distinctions in Human Performance Model", *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. smc-13, pp. 257-266, 1983.
15. Rasmussen, J. "The role of hierarchical knowledge representation in decisionmaking and system management", *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. smc-15, pp. 234-243, 1985.
16. Schattkowsky, T., Lohman, M. "Towards employing UML model mappings for platform independent user interface design", *Proceedings of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces* (October 2, 2005, Montego Bay, Jamaica), 2005.
17. Schmidt, D.C. "Model-driven engineering", *IEEE Computer*, Vol. 39, No. 2, 2006, pp. 25-31.
18. Schneiderman, B. "The future of interactive systems and the emergence of direct manipulation", *Behaviour and Information Technology*, Vol. 1, pp. 237-256, 1983.
19. Sottet, J.-S., Clavary, G., Favre, J.-M. "Ingénierie de l'interaction homme-machine dirigée par les modèles", *Actes des 1ères journées sur l'Ingénierie Dirigée par les Modèles IDM'05* (June 30-July1, 2005, Paris), 2005, pp. 67-82.
20. Tariq, N. A., Akhter, N. *Comparison of Model Driven Architecture (MDA) based tools*, Royal institute of Technology (KTH), Karolinska University Hospital, Stockholm, 2005.
21. Vanderdonckt J. "A MDA-compliant environment for developing user interfaces of information systems", *Advanced Information Systems Engineering: 17th International Conference (CAiSE 2005)*, pp. 16-31, 2005.
22. Vicente, K. J. *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*. Laurence Erlbaum Associates, 1999.
23. Vicente, K. J. "Ecological interface design: progress and challenges", *Human Factors*, Vol. 44, pp. 62-78, 2002.
24. Vicente, K. J., Rasmussen, J. "Ecological interface design: theoretical foundations", *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. smc-22, pp. 589-606, 1992.
25. Wong, B.L. William "The cognitive engineering – design gap". *Ninth Australian Conference on Computer-Human Interaction (OzCHI'99).* J. Scott and B. Delgarno (eds), CHISIG, Ergonomics Society of Australia, Charles Sturt University, Wagga Wagga, Australia, pp. 196-198. 1999.