

Editor Definition Language and Its Implementation

Audris Kalnins, Karlis Podnieks, Andris Zarins, Edgars Celms, and Janis Barzdins

Institute of Mathematics and Computer Science,
University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
{audris, podnieks, azarins, edgarsc, jbarzdin}@mii.lu.lv

Abstract. Universal graphical editor definition language based on logical metamodel extended by presentation classes is proposed. Implementation principles of this language, based on Graphical Diagramming Engine are described.

1 Introduction

Universal programming languages currently have become more or less stable, the main effort in this area is oriented towards improving programming environments and programming methodology. However, the development of specialised programming languages for specific areas is still going on (most frequently, this type of languages is no more called programming languages, but specification or definition languages). One of such specific areas is the definition of graphical editors. The need for various graphical editors and similar tools based on graphical representation of data increases all the time, because due to increased computer speeds and size of monitors it is possible to build graphical representations for wider subject areas. In this paper the **Editor Definition Language (Eddl)** for a simple and convenient definition of wide spectrum of graphical editors is proposed, and the basic implementation principles of Eddl are described.

Let us mention some earlier research in this area. Perhaps, the first similar approach has been by Metaedit [1], but its editor definition facilities are fairly limited. The most flexible definition facilities (and some time ago, also the most popular in practice) seem to be the Toolbuilder by Lincoln Software. Being a typical meta-CASE of early nineties, the approach is based on ER model for describing the repository contents and special extensions to ER notation for defining derived data objects which are in one-to-one relation to objects in a graphical diagram. The diagram itself is being defined by means of a frame describing graphical objects in it and the supported operations. A more academic approach is that proposed by Kogge [2], with a very flexible, but very complicated and mainly procedural editor definition language. Another similar approaches are proposed by DOME [8] and Moses [9] projects, with fairly limited definition languages. Several commercial modelling tools (STP by Aonix, ARIS by IDS

a *dependency* between business activities (this dependency may be a *message* passed from one activity to another, but also something more general). An activity may be triggered by an (external) *business event*. Business activity may have a *performer* — a *position* within a company. This example will be used in the paper to demonstrate the editor definition features. Fig.1 needs one technical remark to be given. Attributes of a class there are extracted as separate classes, linked via an association with the predefined role name **has attribute** to the parent class. This attribute extraction is technically convenient for defining the presentation language part. Otherwise the logical metamodel is an arbitrary UML class diagram.

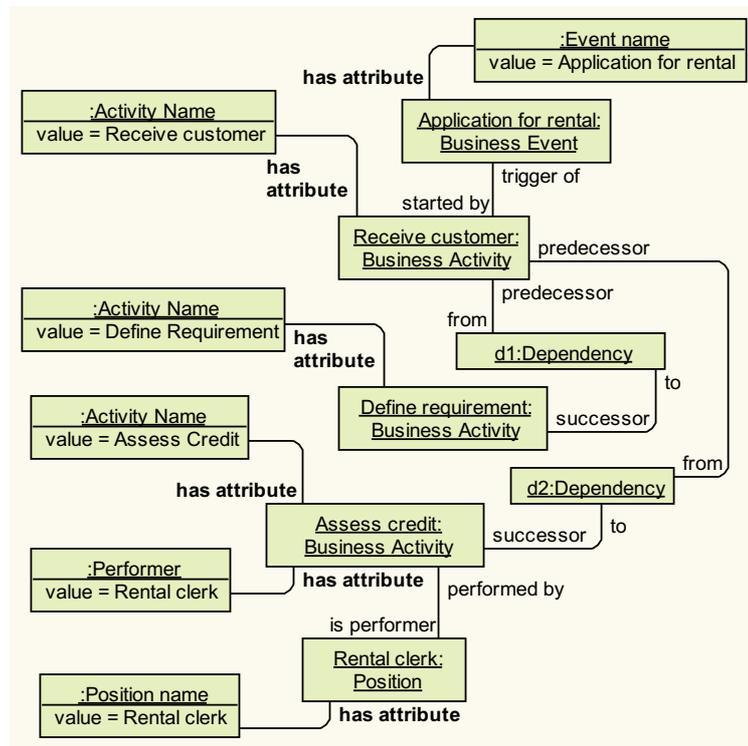


Fig. 2. Example of instance diagram

Fig.2 shows an example of an instance diagram (object diagram in UML terms) corresponding to the logical metamodel in Fig.1. Our goal is to define the corresponding editor, which in this case could be able to present the instance diagram as a highly readable graphical diagram in Fig.3 (where the traditional rendering of dependencies by oriented lines is used). A special remark should be given with respect to *Position*. It is not explicitly represented in diagram in

Fig. 3, but double-clicking on the performer name is assumed to **navigate to** (i.e. to open) a special editor showing the relevant position (this other editor is not specified here). The **navigation** and the **prompting** (the related action by means of which such a reference can be easily defined) are integral parts of our editor definition facilities.

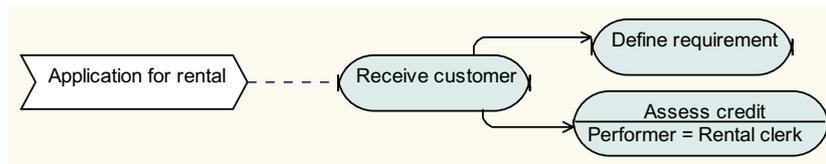


Fig. 3. Business activity diagram example

Roughly speaking, the goal of our definition language is to describe the translation of pictures like Fig. 2 into equivalent pictures like Fig. 3. So it is a sort of graphics translation language.

Now let us start a detailed outline of the EDDL. Like any real language, it contains a lot of details, therefore we will concentrate only on the basic constructs. The language will be presented as an extension of the logical metamodel adhering to the UML class diagram notation. Fig.4 demonstrates the use of EDDL for the definition of the example editor (with some minor details omitted). In this figure rectangles represent the same classes from the logical metamodel in Fig. 1, but rounded rectangles represent classes being the proper elements of EDDL. Classes with class names in bold represent abstract classes, which cannot be modified (they are used mainly for inheritance). Similarly, bold role names of associations represent the fixed ones. We remind that the underlined attributes (to be seen in EDDL classes) are class attributes (the same for all instances) according to UML notation.

The first element added to the logical metamodel is the **diagram** class (*Business activity diagram*), together with standard associations (with the role name **contains**) to the contained diagram objects, and as a subclass of the fixed *Diagram* class. One more standard association for diagram is the refinement association (**refines**), which defines that a *Business Activity* can be further refined by its own *Business activity diagram* (this definition is sufficient for the Editor Engine to enable this service).

Each of the metamodel classes, which must appear as graphical objects in the diagram, are linked by an unnamed association to its **presentation class** — a subclass of standard classes **box** or **line**. The presentation class may contain several class attributes (with predefined names). Thus the presentation class for *Business Activity* — the *Activity box* class says that every business activity must be represented by a rounded rectangle in a light blue default colour. The *Icon* representing this graphical symbol on the editor's **symbol palette** (to create a new business activity in a diagram) is also shown. For presentation classes

being lines the situation is similar, but there may be lines corresponding to associations in the metamodel (*Triggering line*) or to classes (*Dependency line*). The latter case is mostly used for lines having associated texts in the diagram (corresponding to attributes of the class; here the *Message name*). For showing the direction of line (and other similar features) the relevant role names from the metamodel are referenced in the presentation class (e.g. **start**=*predecessor*).

Class attributes are being made visible in diagrams by means of a **compartment** presentation class. The most important attribute of a compartment class is the **position** — for boxes in the simplest case it means the number of the horizontal slice of the box, for lines it means the relative positioning (start, middle, end). Compartment class may contain also style attributes (visible **tag**, separator, font, etc.).

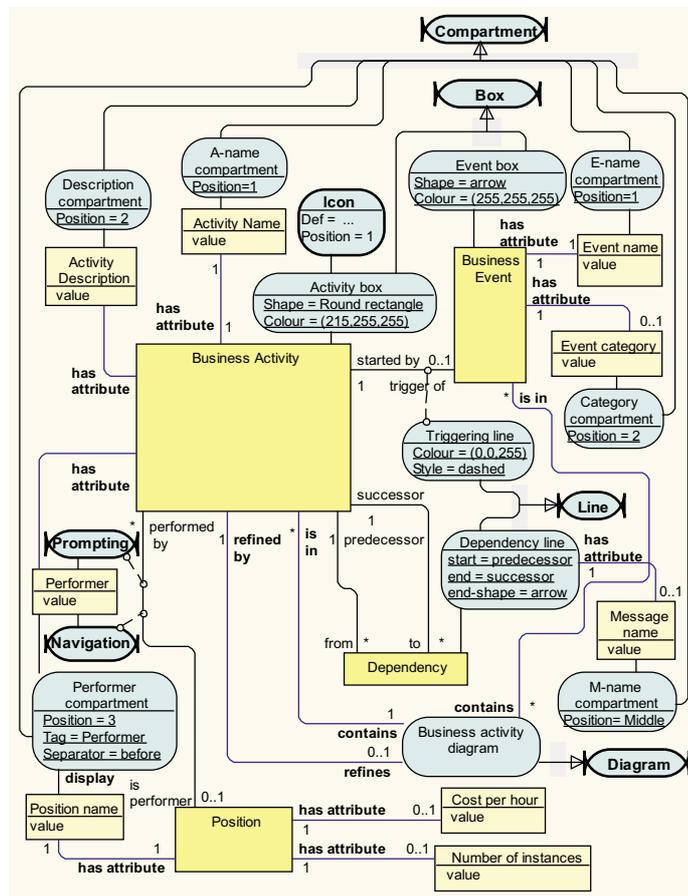


Fig. 4. Business activity diagram editor definition in EdDL

The most interesting element in this EDDL example is the definition of **prompting** and **navigation**. They are both related to the situation when an attribute (i.e. its value) of a metamodel class (the *Performer* for *Business activity* in the example) actually is determined by an association of this class (a derived attribute in UML terms). Here the *Performer* value actually must be equal to the *Position name* of that *Position* instance (if any) which is linked by the association having the role name *Performed by* (the fact that a *Position* must be represented by its *Position name* is defined by the display association). Prompting here means the traditional service found in an editor that a value can be selected from the offered list of values (value of *Performer* selected from the list of available *Position names*), with the automatic generation of the relevant association instance as a side effect. The navigation means the editor feature that double-clicking on the *Performer* field (which presents the *Performer* attribute) in a *Business activity* box automatically invokes some default editor for the *Position* (the target of the association). Both *Prompting* and *Navigation* are shown in the EDDL as fixed classes linking the attribute to the relevant association (they may have also their own attributes specifying, e.g. the prompting constraints). Note that *Position* has no presentation class in the example, consequently its instances are not explicitly visible in the *Business activity diagram*.

Certainly EDDL contains more features than demonstrated in Fig. 4, e.g. various uniqueness constraints, definitions for attribute “denormalisation”, modes of model/diagram consolidation etc. The EDDL coding shown in Fig. 4 was simplified to make it more readable. The actual coding used for the commercial version of EDDL is much more compact, here the metamodel class attributes are defined in the traditional way, and most of Presentation classes are coded just as UML constraints (properties) inside a metamodel class. Nevertheless the semantics of this language is just the one briefly described in the paper. We assert that EDDL is expressive enough to define practically any types of editor that could be used to build related sets of diagrams in system modelling area. Namely the inter-diagram relations such as prompting and navigation are the most complicated ones, and they successfully managed in EDDL. Finally, Fig. 5 shows the defined editor in action.

3 EDDL Implementation Principles

EDDL has been implemented by means of an interpreter which in this case is named **Editor Engine**. When an editor has been defined in EDDL the Editor Engine acts as the desired graphical editor for the end user. Here only the main principles of implementation will be discussed. The first issue is the information storage. It is universally accepted nowadays that the logical metamodel describes directly (or very close to it) the physical structure of the tool repository. This repository can be an OODB, a special tool repository (e.g. Enabler [6]) or a relational DB. This is one more argument why the editor definition should be based on a separately defined metamodel.

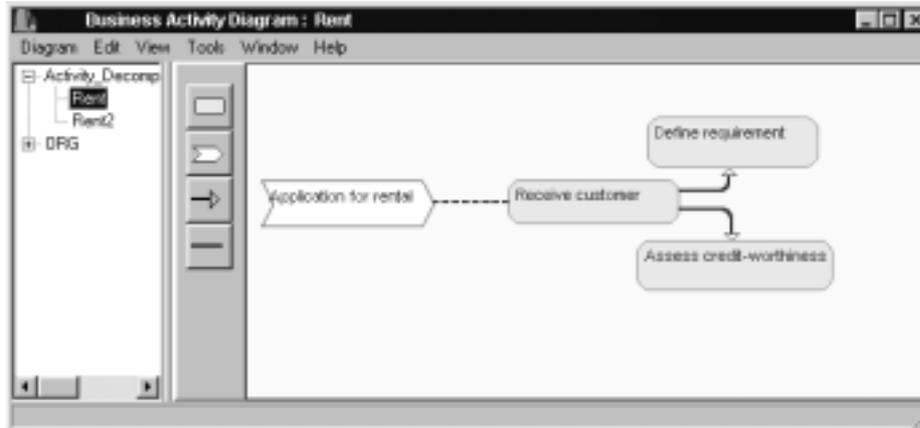


Fig. 5. Editor in action

Fig. 6 shows the general architecture of the EdDL approach. A key aspect is that Editor Engine (EE) relies on **Graphical Diagramming Engine (GDE)** for all diagram drawing related activities. The primitives implemented by GDE — diagram, box, line, compartment etc. and the supported operations on them are very fit for this framework.

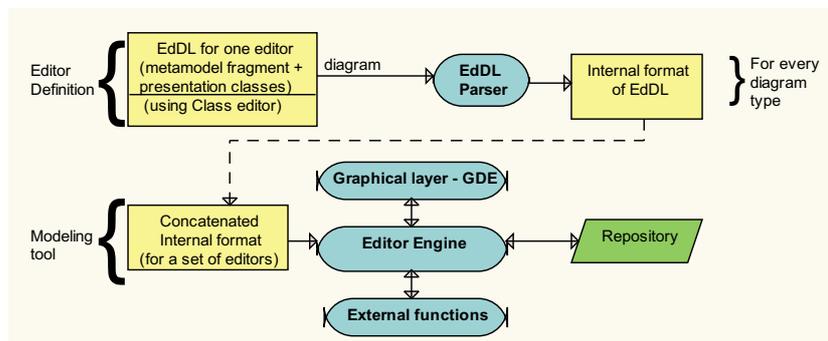


Fig. 6. Architecture of EdDL implementation

Thus the interface between EE and GDE is based on very appropriate high level building blocks, there is no need for low level graphical operations in EE at all. The GDE itself was developed by IMCS UL initially within the framework of ADDE project, with a commercial version later on. It is based on very sophisticated graph drawing algorithms [7].

The general technology of using EdDL for defining a set of editors is quite simple. For each of the editors its definition on the basis of the relevant meta-

model fragment is built. Then these definitions are parsed and assembled into one set. EE “performs” this set, behaving as a modelling tool containing the defined set of diagrams. A lot of tool functionality — “small operations” such as copy-paste and “large operations” such as load and save are implemented in EE in a standard way and need no special definitions. Certainly, external modules can be included for some specific operations.

The practical experiments on using EdDL and EE have confirmed the efficiency and flexibility of approach. The defined editors (like the one in Fig.5) behave as industrial quality graphical editors. The flexibility has been tested by implementing full UML 1.3 and various specific business process related extensions to it.

4 Conclusions

The paper presents a brief outline of the graphical editor definition language EdDL and its implementation. But we can view all this also from a different angle. Actually a new kind of metamodel concept application for a specific area — editor definition — has been proposed. However this approach can be significantly more universal, since it is generally accepted that object model (Class diagram) is a universal means for describing the logical structure of nearly any system. Thus the same approach of extending this model by special “presentation” classes could be used, e.g. to define model dynamics, simulation etc., but this is out of scope for this paper.

References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: *Metaedit — a flexible graphical environment for methodology modelling*. Springer-Verlag (1991)
2. Ebert, J., Sutzenbach, R., Uhe, I.: *Meta-CASE in Practice: a Case for KOGGE*. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Catalonia, Spain (1997)
3. ESPRIT project ADDE. <http://www.fast.de/ADDE>
4. Sarkans, U., Barzdins, J., Kalnins, A., Podnieks, K.: *Towards a Metamodel-Based Universal Graphical Editor*. Proceedings of the Third International Baltic Workshop DB&IS, Vol. 1. University of Latvia, Riga, (1998) 187-197
5. Rumbaugh, J., Booch, G., Jacobson, I.: *The Unified Modeling Language Reference Manual*. Addison-Wesley (1999)
6. *Enabler Concepts Release 3.0*, Softlab GmbH, Munich (1998)
7. Kikusts, P., Rucevskis, P.: *Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors*. Lecture Notes in Computer science, Vol. 1027. Springer- Verlag (1996) 361-364
8. *DOMe Users Guide*. <http://www.htc.honeywell.com/dome/support.htm>
9. Esser, R.: *The Moses Project*.
<http://www.tik.ee.ethz.ch/ mooses/MiniConference2000/pdf/Overview.PDF>