# A DECLARATIVE EXECUTABLE MODEL FOR OBJECT-BASED SYSTEMS BASED ON FUNCTIONAL DECOMPOSITION

Pierre Kelsen
*Laboratory for Advanced Software Systems*
*University of Luxembourg*

Keywords: Declarative models, executable, object-oriented programming, functional programming, software complexity, functional decomposition.

Abstract: Declarative models are a commonly used approach to deal with software complexity: by abstracting away the intricacies of the implementation these models are often easier to understand than the underlying code. Popular modeling languages such as UML can however become complex to use when modeling systems in sufficient detail.
In this paper we introduce a new declarative model, the EP-model, named after the basic entities it contains - events and properties - that possesses the following features: it has a small metamodel; it supports a graphical notation; it can represent both static and dynamic aspects of an application; finally, it allows executable models to be described by annotating model elements with code snippets. By leaving complex parts at the code level this hybrid approach achieves executability while keeping the basic modeling language simple.

## 1 INTRODUCTION

Abstraction is a key concept for dealing with complexity. By abstracting away details of the implementation one can construct a higher-level model that is easier to understand than the underlying code. Although the relations between successive abstraction layers are varied, a common theme is that of separating what a system does from how it is actually done. We call the approaches that rely on this distinction declarative.

An important element of a declarative approach is the language used for representing the high-level models. The de-facto standard for modeling object-oriented systems is the Unified Modeling Language (Object Management Group, 2003). The UML is a powerful language for describing systems at various levels of abstraction and from multiple viewpoints. It has a large number of diagrams available for describing systems from different perspectives, each with their own syntax and semantics. This expressiveness also means that UML is a rather large and complex language (Kobryn, 2002; Siau and Cao, 2001).

The complexity and size of the language becomes a hindrance when designing systems at a detailed level. While it is possible in principle to transform UML into an executable language (Raistrick et al., 2000) by instrumenting it with a precise Action Semantics (Alcatel et al., 2000) this results in an even bigger language. Indeed executability and simplicity seem to be conflicting goals if we judge by previous attempts. The main subject of this paper is a new executable model, the EP-model, that is based on a rather trivial observation: certain aspects of programs can be easily presented in a simple form at a declarative level while other aspects are much more difficult to capture at such a level. Our basic approach to this problem is that of leaving things that are truly complex to describe at a low level (source code) and extracting only those aspects that can easily be presented.

We now discuss the main features of EP-models and contrast them with existing approaches. The simplicity of EP-models is mainly due to the small number of concepts that they are based on: indeed the high-level metamodel can be described using only two types of entities - events and properties - and four types of relationships among those entities.

The second main feature of EP-models is their executability. Executability by itself is not a new idea (e.g., (Belina and Hogrefe, 1989; Raistrick et al., 2000)). What makes our model interesting is the fact that executability is achieved without relying on a

Kelsen P. (2006).
A DECLARATIVE EXECUTABLE MODEL FOR OBJECT-BASED SYSTEMS BASED ON FUNCTIONAL DECOMPOSITION.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 63-68
DOI: 10.5220/0001317300630068
Copyright © SciTePress

overly complex language for the modeling notation. Instead we propose a hybrid approach in which the model itself is unchanged but code segments annotate the various modeling elements to allow executability. A useful characteristic of our hybrid approach is the ”locality” of the code segments: indeed each code snippet can only refer to the model elements that are adjacent to the element that it annotates. Clearly this locality reduces coupling since it disallows the code to access elements that it is not related to. Although there have been a few approaches to reduce coupling at the method level (the Law of Demeter (Lieberherr and Holland, 1989) is representative of such approaches) current approaches are rather low-level in the sense that they refer to an existing class structure. On the other hand the EP-models provide a ”sandboxing” approach for code that is situated at a higher semantic level.

Finally, EP-models model both static and dynamic aspects of a system in a single diagram. On the other hand UML separates static and dynamic aspects into different diagrams. One reason for this difference lies in the fact that while UML is largely grounded in the object-oriented paradigm our model combines ideas from both object-oriented and functional programming: it borrows the notion of state from object-oriented programming while representing dynamic behavior as functions without side-effects that are decomposed over the state. We remark that the idea of combining the functional and object-oriented paradigms is not new but most attempts have focused so far at the level of programming language design (e.g., (Hughes and Sparud, 1995; Rémy and Vouillon, 1997; Odersky and Wadler, 1997)).

## 2 AN EXAMPLE: FLASHCARDS

To illustrate the concepts introduced in this paper, we will make use of a simple application called *FlashCards* that will be used as a running example. The application allows the user to design and work with a set of flash cards. A flash card contains of a question and an answer. Flash cards are commonly used as a study aid. The application should allow the user to add a number of flash cards, specifying for each card the corresponding question-answer pair. The main window should present an overview of the cards entered so far. The user can enter a special *quiz* mode: in this mode he can review the flash cards one at a time. For each flash card the question is displayed and the user can choose to also view the answer.

## 3 THE STATIC VIEW: LOCAL PROPERTIES AND THE SYSTEM STATE

The static structure of an EP-system is given by a set of **local properties** in each model. (A second class of properties named *query properties* will be introduced in section 6). Local properties have a name and a **type**. We shall assume that no two properties in the same model have the same name. The type has a name and an associated set of values. This type can be either **internal** or **external**: an internal type is given by another model in the EP-system. Examples of external types are the built-in types of a programming language or a class in a class library; external types are not represented by EP-models. A property is either single-valued or multi-valued. We call multivalued properties also **collection properties**.

**Example 1** *We name the EP-model for representing a flash card* FlashCard*. This model contains two properties, named* question *and* answer*, of type* java.lang.String*, an external Java type. Another example is the* Main *EP-model representing the main window of the FlashCards application. The* addButton *and* quizButton *properties are properties of an external type (SWING components). The other properties of the Main model -* list, cardDialog, quizDialog, flashCards *- have an internal type represented by an EP-model. We note that the* flashCards *property is a collection property of type* FlashCard *- this property refers to the collection of flash cards entered by the user.*

When an EP-model executes, it goes through a series of *system state*s. Informally, a system state is a set of instances, each instance belonging to some model and assigning concrete values to the local properties in that model.

For a more formal definition of a system state:

**Definition 1** *A **valuation** for a model $M$ is a function that assigns to each local property $p$ in $M$ a value of the type of $p$.*

**Definition 2** *An **instance** of model $M$ is a triplet $(M, id, \phi)$ where $M$ is a model, $id$ is a* name *for the instance and $\phi$ is a valuation for $M$. We call $\phi(p)$ is the* value *of (local) property $p$ in $M$ on this instance.*

**Definition 3** *A **system state** is a set of instances.*

**Condition** In any system state the id’s of the instances are unique.

## 4 EVENTS, THE TRANSFORMATION MAPPING AND CENTERED FUNCTIONS

External triggers that modify the current system state are represented in a model by **local events**. (Another class of events - remote events - will be presented in a later section.) A local event has a name, a **type** and a **source.** We shall assume that no two events in the same model have the same name.

The event type is platform-specific: in Java an event type is a pair $(l, m)$ where $l$ is a listener interface and $m$ a method of this interface.

The source of an event is a local property in the model that contains the event.

**Example 2** *In the* Main *model we have two local events* add *and* quiz. *The* add *event represents pressing the add button and the* quiz *event occurs when we press the quiz button. Both events have as type* (java.awt.event.ActionListener, actionPerformed). *The source of the* add *event is the* addButton *property of the* Main *model and the source of the* quiz *event is the* quizButton *property of the* quiz *event.*

**Definition 4** *We say that a **local event occurs on an instance** $x$ if an (external) event of the given type occurs on the object referred to by the source of the local event. In this case we also say that instance $x$ is the **locus** of the local event.*

**Example 3** *When we press the* add *button in the main window, an event occurs on the* Main *instance; this instance is then the locus of this event.*

**Definition 5** *When a local event occurs on instance $x$ of the **current system state**, then the current state is replaced by a new state which we call the **result state**.*

**Definition 6** *For a given local event $e$ the mapping that associates with each system state and instance of this state on which e occurs a result state is called the **transformation mapping for e** and is denoted by $F_e$.*

Mathematically we describe transformation mappings using *centered functions.*

**Definition 7** *A **centered state** is a pair $(s, x)$ where $s$ is a system state and $x$ is an instance of $s$. We call $x$ the **center** of the centered state.*

**Notation** We also use $s(x)$ to denote a system state $s$ centered at $x$.

**Definition 8** *A **centered function** is a function whose domain is a set of centered states (for the given EP-system).*

We may view the transformation mapping $F_e$ as a centered function that maps the current state centered at the locus of the event to the result state.

The transformation mapping completely describes the dynamic behavior of an EP-system. Indeed if the EP-system expresses the transformation mapping precisely, then the EP-system is executable. The remainder of this paper is essentially looking at the question of how to best represent centered function $F_e$ at the level of the EP-models.

## 5 BICENTERED FUNCTIONS

To represent the transformation mapping, we shall decompose it into simpler functions. First we need to define the effect an event has on a system state.

**Definition 9** *A **local event** $e$ **affects a local property** $p$ if for some system state the value of this property is changed on some instance of this state when the event occurs. In this case we also say that the local event **affects property** $p$ **on this instance**.*

**Example 4** *The effect of the* quiz *event in the* Main *model is to show the QuizDialog, to set the questionField (a text field) to the first question and to set the* index *property indicating the position of the current card among the stack of flash cards. Thus the* quiz *event affects the* visible *and* index *properties of* QuizDialog.

To fully describe a local event $e$, it suffices to specify the effect of $e$ on each local property affected by $e$. The effect of $e$ on property $p$ can be expressed by the function that returns the new value of property p on an instance of the result state after $e$ occurs on the current state; we denote this function by $F_{e,p}$.

**Example 5** *Let $e$ denote the* quiz *event in the Main model and let $p$ stand for the* visible *property in the* QuizDialog *model. Then $F_{ep}$ represents the new value of the* visible *property when the* quiz *event occurs; in this case $F_{e,p} = true$.*

The value of $F_{e,p}$ depends on *two* centered states:

- the current state $s(x)$ centered at the locus of local event $e$, i.e., at the instance where $e$ occurs
- the result state centered at an instance at which we are evaluating the new value of $p$

This dual dependency motivates the next definition.

**Definition 10** *A **bicentered function** is a function of the form $f(s(x), s'(y))$ where $s(x)$ and $s'(y)$ are two system states centered at instances $x$ and $y$, respectively.*

We note that function $F_{e,p}$ is a special bicentered function where the second argument state is the result state.

The centered function $F_e$ is fully specified by the functions $F_{e,p}$, where $p$ ranges over all properties affected by $e$. We have thus reduced the problem of decomposing the centered transformation mapping $F_e$

into that of decomposing the related bicentered functions $F_{e,p}$. Before we address the decomposition of bicentered functions, we explain how to decompose centered functions since they will be used in the decomposition of the bicentered functions.

## 6 DECOMPOSING CENTERED FUNCTIONS USING PROPERTY GRAPHS

In this section we shall describe how to decompose centered functions and how to represent this decomposition in EP-models.

The computation of a centered function will be based on the decomposition of this function into "simpler" functions. Each centered function is represented at the model level by a **query property**. Just like local properties query properties have a *name* and a *type*. Local properties and query properties together make up the set of properties of an EP-system. To decompose the query property, we first describe which values a query property depends on. This is done by defining for each query property a *property graph.*

**Definition 11** *The **property graph** for a query property $q$ is defined as follows: the set of nodes is a set of properties that contains property $q$ and other local or query properties. There are three types of edges: **local edges**, **forward edges** and **inverse edges**. A local edge is given by a pair $(p_1, p_2)$ of properties in the same model. Forward edges and inverse edges are labeled by a property $p$ which we call the **link property**; for forward properties the link property is a property in the model of $p_1$ whose type is a model containing $p_2$ while for inverse properties the link property belongs to the model of $p_2$ and its type is a model containing $p_1$. The link property is a local property or a query property.*

Intuitively, a local edge represents a dependency of two properties on the same instance while forward and inverse edges represent a dependency between two properties on two separate instances connected by the link property $p$.

At the model level we represent the property graph by adding a *parent* relationship link from a query property to each of its children properties. The parent relationship has two attributes: the link property (undefined for local edges) and type (local/forward/inverse).

**Example 6** *Figure 1 shows the property graph for the query property* nextIndex *in the* QuizDialog *model: this query property computes the index of the next card to be displayed in the quiz dialog. The property graph contains two local edges and one inverse edge (having link property* quizDialog*). The* flashCards *and* index *properties are local properties.*

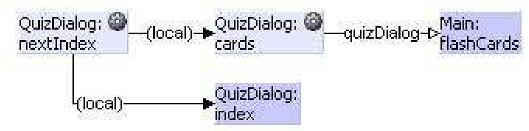

Figure 1: Property graph for *nextIndex.*

We add a code snippet to each query property that computes the value of the query property in terms of the values of children properties.

**Example 7** *The code snippet that computes the value of property* nextIndex *is given below. Note that it only uses values of properties that are children of itself in the event graph (see figure 1).*

```
if (index<cards.size()-1)
     result = index+1;
  else
     result = 0;
```

## 7 DECOMPOSING BICENTERED FUNCTIONS USING EVENT GRAPHS

To decompose the $F_{e,p}$ functions, we will need to precisely define what instances in the result state are affected by an event. This will be done by associating with each local event an *event graph*.

To define the event graph for a local event, we first add to each model a set of **remote events**. Remote events have a name but unlike local events they do not have a type and source attribute. Local events and remote events together make up the set of events of an EP-system. We may think of remote events as the representatives of a local event in other EP-models.

**Definition 12** *The nodes of the **event graph** of a local event comprise the local event as well as a set of remote events. The edges of the event graph are either* forward,inverse *or* local *edges. A **forward edge** $(e_i, e_j)$ is labeled by a property $p$ in the model of $e_i$; $e_j$ must be an event in the model that is the type of $p$. The forward edge is denoted by $e_i \rightarrow_p e_j$. An **inverse edge** $(e_i, e_j)$ is labeled by a property $p$ in the model of $e_j$; $e_i$ must belong to the model that is the type of $p$. The inverse edge is denoted by $e_i \hookrightarrow_p e_j$. A **local edge** $(e_i, e_j)$ connects two events in the same model and does not carry a label; it is denoted by $e_i \rightarrow e_j$. For foward and inverse edges we call property $p$ the **link property**. The link property is a local property or a query property. We call the edges in the event graph also **event links**.*

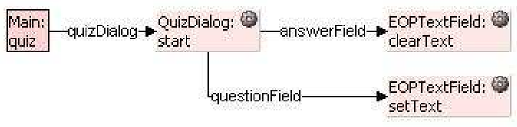

Figure 2: Event tree of the *quiz* event.

**Notation** We denote the event graph for a local event $e$ by $G_e$.

At the level of EP-models we represent event links using a parent relationship between events: each parent link connects a source event to a target event; it has as attributes a *property* (the link property, undefined for local links) and a *type* attribute (with values: forward/inverse/local).

**Example 8** *Figure 2 shows the event graph of the* quiz *event in the* Main *model. All events but the* quiz *event are remote. All event links are forward links. Three models are involved:* Main, QuizDialog *and* EOPTextField, *the latter model representing a text field. This reflects the fact that the* quiz *event affects properties in the* QuizDialog *instance but also sets the contents of the question and answer text fields which are modeled by* EOPTextField.

To decompose the functions $F_{e,p}$ over the event graph, we define auxiliary functions on the nodes of the event graph. These functions compute and transmit the information required by the $F_{e,p}$ function.

**Definition 13** *A* ***parametrization*** *of an event graph $G_e$ is given by*
*- attaching to each event of $G_e$ a set of centered functions represented by query properties*
*- assigning to each remote event of $G_e$ a set of* ***parameters****, where each parameter has a* name *and a* type
*- assigning to each event link $l$ and to each parameter $g$ in the target event of $l$ a function $F_{g,l}$ that expresses the parameter $g$ in terms of parameters and centered functions $f_i$ at the source of $l$: $g = F_{g,l}(f_1, \dots, f_k)$.*

We now describe how the parametrization is represented at the model level. We may view a local property as a very simple centered function that returns the value at the center of its argument state. We represent more general centered functions in EP-models by query properties (see previous section).

We represent the attachment of a centered function to an event by introducing a *feeds* relation from the query property to the event. Each non-local event in an EP-model has a set of *parameters*. The function $F_{g,l}$ is represented by attaching a code snippet for parameter $g$ to the event link from the source to the target event.

We have now all the pieces together for implementing a local event $e$ at the level of an EP-system: decompose the transformation mapping $F_e$ into the bi-centered functions $F_{e,p}$, one for each property $p$ affected by $e$. For each function $F_{e,p}$ create an event graph with a parametrization that provides a functional decomposition of $F_{e,p}$. Note that this may entail creating new query properties feeding remote events. Finally decompose these query properties using property graphs. Repeating these steps for each local event will result in an executable EP-system that represents the final application. For a more detailed description and additional examples the reader is referred to (Kelsen, 2006).

## 8 APPLICATIONS OF EP-SYSTEMS

We see three directions for future work that correspond to potential applications of EP-models.

1. Modeling applications: as we have seen in this paper we can use EP-models to model simple applications. Because EP-models are executable the EP-system in fact constitutes the application: no additional code is needed. Of course in practice one would write a code generator for efficient execution. To prove the feasibility of this approach we have developed a tool (Glodt and Kelsen, 2006) that provides a visual environment for designing EP-systems: the tool is implemented as an Eclipse plug-in that supports editing and executing EP-models with rule-based background code generation. It is not clear yet whether EP-models are a reasonable approach for modeling large applications: to answer this question, we are planning to develop such an application using our tool. In any case EP-models would supplement rather than replace existing UML models: indeed many UML artifacts such as use cases and deployment diagrams could supplement EP-models by providing high-level views of the application and also describing aspects not represented by our models.

2. Mastering software complexity: EP-models have a number of features that may help in controlling the complexity of the resulting system: first EP-systems exhibit *locality* because code snippets may only depend on values that are located on ”adjacent” elements in the EP-system. We have implemented (Glodt and Kelsen, 2006) this locality using a sandbox model: the sandbox for a code snippet only contains the values that are accessible by this code in the model. This locality should help in reducing coupling ((Stevens et al., 1999)) in the resulting application. Second the models provide

facilities for comnprehending the dynamic behavior of an EP-system: we can understand the effect of an event on the system by following the edges of the event tree. Similarly data dependencies can be quickly discovered with the help of property trees.

3. A laboratory for testing object-oriented methods and concepts: since our models provide a restricted environment for describing the static and dynamic aspects of an application, they should be easier to analyze and can be used as a testbed for developing mathematical models that may carry over to more unrestricted environments. For example techniques such as design patterns (Gamma et al., 1995) or refactoring (Fowler, 1999) could be examined in these more restricted models. This could potentially provide a more rigorous basis for these techniques that could carry over at least in part to more traditional software programs. Another benefit of trying out these techniques on EP-models is of course their potential to make the EP-modeling process more effective.

## 9 CONCLUSIONS

We have presented a declarative model, named the EP-model. EP-models are based on a small metamodel comprising two types of entities, *events* and *properties*, and four binary relationships between those entities.

EP-models are executable; executability is achieved by associating code snippets with entities and relationships. These code snippets compute functions without side-effects. This hybrid approach allows one to keep the basic modeling language simple by leaving complex parts at the code level. The code snippets obey a locality constraint: they can only use values connected with neighboring modeling elements. This reduces the amount of coupling in the resulting application.

EP-models combine static and dynamic aspects of a system in a single diagram. They combine the notion of state from object-oriented programming with the notion of functional decomposition from functional programming.

Future work will examine

- whether EP-models can be used to model applications of a realistic size and what the advantages/disadvantages are over existing UML-based methods;
- whether EP-models can be used as a cleanroom for testing object-oriented ideas and concepts. As an example we plan to study refactoring and design patterns in the context of these models. Because of the simple structure and executability of these models, such a study could provide a more rigorous basis for some of these techniques which could then possibly be carried over to more traditional software programs.

## REFERENCES

Alcatel, I-Logix, Kennedy-Carter, Technologies, I. K., Technology, I. P., Corporation, R. S., and AB., T. (2000). Action semantics for the UML. In *Document ad/2001-03-01. OMG*.

Belina, F. and Hogrefe, D. (1989). The ccitt-specification and description language sdl. *Comput. Netw. ISDN Syst.*, 16(4):311–341.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. FOW m 01:1 1.Ex.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley.

Glodt, C. and Kelsen, P. (2006). Demos: A tool for declarative executable modeling of object-based systems.

Hughes, J. and Sparud, J. (1995). Haskell++: An Object-Oriented Extension of Haskell. In *Proceedings of Haskell Workshop, La Jolla, California*, YALE Research Report DCS/RR-1075.

Kelsen, P. (2006). A declarative executable model for object-based systems based on functional decomposition. Technical Report TR-LASSY-06-06, Laboratory for Advanced Software Systems, University of Luxembourg. `http://lassy.uni.lu/demos/documentation/TR_LASSY_06_06.pdf`.

Kobryn, C. (2002). Will uml 2.0 be agile or awkward? *Commun. ACM*, 45(1):107–110.

Lieberherr, K. and Holland, I. (1989). Assuring a good style for object-oriented programs. *IEEE Software*, pages 38–48.

Object Management Group (2003). Unified modeling language, march 2003. version 1.5.

Odersky, M. and Wadler, P. (1997). Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA.

Raistrick, C., Wilkie, I., and Carter, C. (2000). Executable UML (xUML). In *Proceedings 3rd International Conference on the Unified Modeling Language UML*.

Rémy, D. and Vouillon, J. (1997). Objective ML: A simple object-oriented extension of ml. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 40–53, Paris, France.

Siau, K. and Cao, Q. (2001). Unified modeling language: A complexity analysis. *Journal of Database Management*, 12(1):26–34.

Stevens, W. P., Myers, G. J., and Constantine, L. L. (1999). Structured design. *IBM Syst. J.*, 38(2-3):231–256.