e-Informatica Software Engineering Journal, Volume 7, Issue 1, 2013, pages: 35–44, DOI 10.5277/e-Inf130104

Reusable Object-Oriented Model

Jaroslav Žáček*, František Huňka*
*Faculty of Science, University of Ostrava
jaroslav.zacek@osu.cz, frantisek.hunka@osu.cz

Abstract

This paper analysis approaches and possibilities of executive model aimed to MDA approach. The second part of the article proposes guideline to create executive model, describes basic interactions to object oriented approach and shows possibilities of creating a core of executable model in Java programming language. Annotations are used for executive model object extension. Reflection concept is used for model execution and synchronization provides extended Petri net formalism defined in [1]. The model has been tested on LFLC software package developed by IRAFM, University of Ostrava to prove the whole concept.

1. Introduction

In present days model transformations in object-oriented programming are focused to speed and automation. The criteria for transformation are concrete programming language, model expressivity and domain usability. In addition the elevation of abstraction should be applied to make modeling easy and simple. Main advantages of this approach are noticeable during initial analysis of application or when user needs to automate some processing. During key requirement identification the higher abstraction level is needed. Reducing model abstraction concretizes this initial design with transformations. Transformation ends on source code level and model becomes platform dependent. But in any time the user can transform model to higher abstraction level and make necessary changes. All these tasks can be done using automated tools and changes are applied on lower source code level. This approach is very useful in agile programming methodologies and enables very fast model changes. One option is to divide models to different levels of abstraction and make a transformation between them. Model transformation process is described in [2] specifications and it is know as a Model-driven architecture (MDA). MDA is a registered trademark of Object Management group. The MDA architecture was established in 2001. A lot of transformation tools for platform independent model (PIM) to platform specific model (PSM) were developed since 2001. Tools allows to transfer abstract model to concrete using with technologies such as Web Services, EJB, XML/SOAP, CSharp, CORBA and others. In addition another standard established in the past such as MetaObject Facility (MOF), Unified Modelling Language (UML),Common Warehouse Metamodel (CWM) and XML Meta Interchange (XMI) are available for MDA support. MDA architecture consists of 4 layers specified as a M0 – M3 and every layer in this specification represents a different level of abstraction. MOF is used for initial domain identification. MOF is specified as a M3 layer in a MDA specification. This layer is a domain specific language, which is used for metamodel description. By this language user can describe M2 lower layer. We can consider UML as an object-oriented metamodel and Web Services or Petri's nets as a non-object-oriented metamodel. Models based on MDA architectures are not focused on model execution. These models are focused on platform

independent model transformation to platform specific model and changing level of abstraction.

Executable UML (also known as xUML or xtUML) is a part of UML specification and aimed to execution compared to regular UML diagram and offers needed standard extension for execution.

Executable UML is defined by these elements:

– Class diagram – defines classes and interactions between their associations for the domain
– Statechart diagram – defines the states, activities and state transitions for a class instances
– Domain chart – describes a modeled domain and relations to other domains
– Action language – defines the actions or operations that perform processing on elements

In fact the Executable UML is an extension to MDA platform and enables making executive models on M1 level from elements described above. An executive model on a higher level of abstraction is created and this model is transformed to programming language source code, mostly 3rd generation programming language.

A framework called M3 action has been developed to make executive modeling easy. This framework has been transformed to open-source project called MXF (Model eXecution Framework). Framework extends model with so called action scripts, which express model execution semantic. By these extensions user is able to change model quickly without any implementation or compilation.

2. Problem formulation

Basic formulation of executive modeling has been described in introduction. As a context of problem we consider MDA architecture on Fig. 1. A bottom layer contains data and is an instance of M1 layer, which creates a model. There is no execution on M0 layer because M0 contains data with no context and therefore higher abstraction to express interactions. Interaction between data is realized on M1 layer, where the classes and their relationships are described. These relationships realize method calling. Fast relationship changing is suitable for modeling. By the thought of changing relationship means change any method calling in any object in the model. Ideally user is able to change relationships and inner class attributes during simulation.

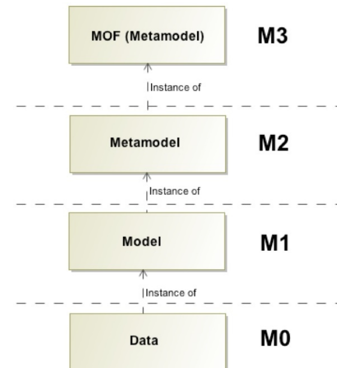

Figure 1. MDA architecture

This execution approach is usually realized on M1 level, which is closed to platform independent model. User can examine classes and their attributes state, make a direct relationship to another class, step the simulation process and ideally read class values in the real time.

2.1. The MDA

The MDA architecture introduced by OMG group was developed to support model-first software development. At first a very abstract model is created, then model is transformed to lower levels of abstraction. Transformations end when model is suitable to generate application source skeleton in corresponding programming language. Automated tools to speed up the process and reduce errors caused by writing code by programmer are available and support this approach.

Model doesn't concern execution and ensures just metadata reflection of workflow. According to [2] MDA is defined by these points:

– Models
– Abstraction
– Platform
– Model Transformation
– The MDA value proposition

MDA specification defines model as a formal specification of functions, structure and system behavior. UML has been chosen as formalism.

According to OMG definition the source code can be concern as a model because this code has a formal specification (all code structures has an exact semantic) and models real machine code, which is available as a program language transformation by compiler or interpreter. This point of view is not interested for object-oriented approach and therefore this model will be considered as a UML model.

Referential model for open distributed processing (MR-ODP) is marked as a suppression of irrelevant detail according to ISO 10746-2 [3]. Model with a high level of abstraction has naturally less detail a posteriori to realization than model with a lower level of abstraction. MDA has been created to start development on a higher level of abstraction and then transform created model to lower level of abstraction until source code is generated. Therefore model drives entire software architecture development.

2.2. Model transformation

MDA generates source code by model transformation. Initial model is platform independent with higher level of abstraction and determined by following points:

- Represents business functionality not tight to technological platform.
- s a detailed model (mostly UML).
- Independent on programming language or operating system.
- Creates baseline to platform-specific model.

PIM is transformed to platform-specified model (PSM) and is adapted to use with target platform. PSM model includes information about business platform and creates PIM mapping to target platform, creates source code skeleton and associated artifacts. As an artifact we can consider deployment descriptor, documentation and build files.

This description implies that we can define a PIM, which can be reusable for different platforms, appropriate PIM to PSM mapping and PSM to source code compiler to target platform as well. In additional this process can by automatized by tools. This transformation is one way only. If the change on lower PSM layer is realized this change cannot be applied on a higher levels in automate process. However, this change could be in a direct conflict with initial modeled purpose. Conversion between PIM to PSM model cannot be realized fully automatically. For example tools cannot determine if account must be marked as an entity EJB or session EJB during the translation.

2.3. The MDA Value proposition

Programming language is an instrument to executive model expressed in UML. This fact has been considered as a disadvantage of model transformation because by this transformation model becomes platform dependent on operating system or specific programming language. Programming language lifetime is limited and when new programming language becomes in use old source code is become useless and must be transferred. Presently using platform independent on operating system approach minimizes the risk of boundedness source code to platform. Using Java technology in these days minimizes boundedness risk. Company's processes are changing and PIM must respond to these changes. The MDA advantage is to preserve high-level views to solve problems – PIM.

2.4. The MDA Execution

In original MDA architecture design was no execution at all. Modeling starts at higher layer and by concretizing model and decomposing (model transformation) the new code is generated. Generated code contains class skeleton. Function interactions between classes are represented by UML relationships only and class itself carries no executive information, instantiation approach or input and output methods. Main disadvantage of this approach is that model cannot use components developed before and model cannot be executed and debugged. PSM to PIM transformation can be made from class diagram (low model view), but this transformation is difficult, cannot be done automatically and for right model identification archetype patterns [2] must be used. To make MDA architecture running automatically an Executable UML extension must be applied.

2.5. M3 Action – Model Execution Framework

M3 action, mostly known as a MXF, is a project focused on executive modeling on a higher level of abstraction (M3). A new language has been defined to describe interactions between elements [4]. Language is based on UML Actions/Activities. From executive point of view, a more abstraction view is available compare to Executable UML. MXF and Executable UML cannot change level of abstraction and creates executable models on a single layer. Metaobject instantiation is performed in M3 abstraction level; therefore tool cannot identify a design pattern of the implementation. Compare to UML the MXF supports aspect-oriented programming due to M3 abstraction level.

One of the main goals of object-oriented programming is reusing components. In all approaches described above there is no mechanism to integrate reusable components to model or make model with reusable components. Approaches discuss creating class skeleton of model in programming language with no direct execution. Model created that way cannot be debugged without changing source code and add some new functionality. MDA architecture is able to generate model from bottom to up (elevate level of abstraction) by using the archetype patterns, but this model lost executive ability by perform this transformation. Executable UML tries to minimize MDA disadvantage by adding more information to object interaction in the model. By applying these techniques an executable model with higher abstraction level from reusable components cannot be created.

3. Defining a new modeling approach

MDA, Executive UML and MXF don't include the requirements to executive model:
- Create model form reusable components.
- Concerning design patterns.
- Flexible change when component is replaced.
- Function and debugging with no code compilation.
- Change the level of abstraction.

These requirements can be realized with minimal generality reduction by extension of object metamodel and applying a reflection tool.

3.1. Reflection

Reflection as a term in information science means ability to read and change program structure and behavior during the program running. Considering to object-oriented programming approach, reflection means ability to read and change object attributes, read and execute the object methods, passing calling results and instantiate new objects. Generally the reflection is able to read object metamodel during program running without changing any object attributes. Reflection is widely used with Smalltalk programming language and scripting languages. Reflection can be used as a universal tool to make object persistent [5] or to generate project documentation.

Reflection enables creating a new object instance entered by name during program running. Following source codes are in Java programming language, but same function can be done with .NET platform and languages defined under Common Language Specification. Basically there are two requirements to programming languages:
- Ability to read object metadata and work with them as a metamodel (object self-identification).
- Some tool to enable object metamodel extension.

The metamodel that carries information about class must be discovered before instantiation.

Fig. 2. shows a representation of metamodel reference. That reference has been found during program running by providing his name – String data type. Execution wrapper is a standalone class. Inner attribute saves metamodel references and instantiated object. For every object is created his instance, special cases as a Library Class are covered by metamodel extension explained in 3.2.2.

At first a constructor must be found to instantiate a class. A simple model has been created for model testing purpose. Model is limited to

```java
public boolean setReflectObjectByName(String name)
        throws ClassNotFoundException, SecurityException,
        NoSuchMethodException, IllegalArgumentException,
        IllegalAccessException, InvocationTargetException,
        InstantiationException {
    boolean instanceLimit = false;
    String instanceMethod = "";
    int pool = -1;
    this.reflectClass = Class.forName(name);
    Annotation[] annotations = this.reflectClass.getAnnotations();
    for (Annotation annotation : annotations) {
        if (annotation instanceof ModelSupport) {
            ModelSupport type = (ModelSupport) annotation;
            if (type.designPattern().equals("Singleton")) {
                instanceLimit = true;
                instanceMethod = type.instanceMethod();
                pool = 1;
            } else if (type.designPattern().equals("Pool")) {
                instanceLimit = true;
                instanceMethod = type.instanceMethod();
                pool = type.pool();
            } else {
                instanceLimit = false;
            }
        }
    }
    if (instanceLimit) {
        return createInstanceLimit(instanceMethod, pool);
    }
    return createInstance();
}
```

Figure 2. Pointer to object metamodel

non-parametric constructor. During instantiation the metamodel is searched and first constructor is called. Result of instantiation is saved to class realizing execution. A method to create instance with no instantiation number limit is described in Fig. 3.

More complex instantiation method is extended by instance count parameter and factory method name. Factory method specification is presented in virtue of factory method name inconsistency [6].

3.2. Analyzing class

Reflection can read all object metadata, all inner attributes, methods, input parameters and return value can be identified. A new class has been created (Fig. 4.) to metamodel verification. Reflection is applied to find the metamodel and all methods are called one by one. Metamodel contains a list of all methods including methods marked as a private by modifier.

Discovered metamodel will be used as an input information to create a graphic model representation. In this graphic representation an order of method calling can be changed if all input attributes and return values have same type.

Eleven modifiers are defined by Java programming language. Modifiers can be characterized as a possible access to objects. In Fig. 3. the modifier is set to private therefore a violation of object-oriented programming is occurred. But it isn't a mistake from instantiation point of view. According to Library Class design pattern, the constructor is defined as an empty constructor therefore Library Class instantiation doesn't change inner state of object. Other classes using a factory method to instantiation requires at least a metamodel extension for factory method identification. Change can be done on graphic model representation level as well. According to reflection all identifiers can be changed, which gives user powerful tool to make changes during the program/model running.

```java
private boolean createInstance() throws SecurityException,
        NoSuchMethodException, IllegalArgumentException,
        IllegalAccessException, InvocationTargetException,
        InstantiationException {

    Constructor[] constructors = this.reflectClass
            .getDeclaredConstructors();
    for (Constructor constructor : constructors) {
        if (!Modifier.isPublic(constructor.getModifiers())) {
            constructor.setAccessible(true);
        }
        this.reflectObject = this.reflectClass.newInstance();
        break;
    }
    return true;
}
```

Figure 3. Basic instantiation method

```java
public boolean invokeAllMethods() throws IllegalArgumentException, IllegalAccessException,
                                                                    InvocationTargetException {
    Method[] allMethods =  this.reflectClass.getDeclaredMethods();
    for (int i=0; i<allMethods.length; i++) {
        Method oneMethod = allMethods[i];
        Class<Type>[] parameters = (Class<Type>[]) oneMethod.getParameterTypes();
        Object[] defaultValues = new Object[parameters.length];
        for (int j=0; j<parameters.length; j++) {
            Type type = parameters[j];
            defaultValues[j] = getDefaultType(type);
        }
        Object returnObj = oneMethod.invoke(this.reflectObject, defaultValues);
        if (returnObj.getClass().isArray())
            System.out.println("Calling method according to type, return value is array.");
        else
            System.out.println("Calling method according to type, return value is " + returnObj.toString());
    }
    return true;
}
```

Figure 4. Invoking methods

If reflection is used to create executive model a question of speed of entire executive lifecycle needs to be consider. Supporting class ensures not just initial instantiation but calling methods and passing parameters as well. According to some sources reflection API is slow. Confirmed by [5] this affirmation is not based on true. By application of Amdahl's law a formula is derived:

$$\text{Slowdown} = \frac{\text{Rtime} + \text{Work}}{\text{Ntime} + \text{Work}}$$

where Rtime is a micro benchmark measurement of a reflection solution and Ntime is a micro benchmark measurement for nonreflective solution. Work is a relative amount and can be interpreted as a Work = Ntime * x , where x is a factor of scaling, which determines time spend with other things. By substitution a formula to slowdown can be derived:

$$\text{Slowdown} = \frac{\frac{\text{Rtime}}{\text{Ntime}} + x}{1 + x}$$

This interpretation of Amdahl's law enables to set referential unit of performance. Rtime/Ntime ratio should be about the same for processors no matter the speed at which the clock is running. After value substitution of instantiation dynamic proxy the ration is equal to 329.4 and slowdown is about 1900% when work is less than 17 times NTime. This numbers seem high but to print "Hello World!" in Java programming language the value of work is equals to 36,000 times Ntime, which a slowdown is under 1%. Therefore there is no noticeable slowdown if reflection technique is applied.

3.3. Class metamodel

According to [7] a metamodel is a domain-specific language oriented towards the representation of software development methodologies and endeavours. After adjusting to class diagram metamodel we can say that metamodeling is an ability to express interactions between classes from metamodel – inner object state. Metamodeling is the act and science of engineering metamodels. Basic metamodel contains information necessary to class representation in concrete programming language.

Two approaches can be use to get metamodel. Model can be obtained from descriptors made before which are tight with created class. This form of implementation is very simple, however descriptor maintenance becomes difficult. When descriptors are defined in high amount the maintenance becomes confusing. If the class doesn't contain descriptors, it cannot be used for metamodel purpose. This type of approach is applied in object-relation mapping known as a Hibernate project.

Second option is use a reflection and read entire object metamodel. This information is obtained during program running and therefore enables dynamic 3rd part library linking with no additional library changes. When class name is provided the reflection interface can read all class attributes, methods, return values and modifiers and pass these values to process on a higher level, typically GUI. In some cases detail information must be known to use class metamodeling. Basic metamodel is not sufficient therefore a new tool for user metamodel extension needs to be found. Reflection must be able to use these extensions during object instantiation and modeling. Annotations are a quiet suitable for user metamodel extension. Annotations are special type of syntactic metadata, which can be add to class source code and extend metamodel expressivity. Metamodel expressivity extension is shown on Fig. 5.

Interface on Fig. 5. is implemented by a class and extends user description part. Reflection allows reading these values and directly decides during program running. In this case a definition of instance is presented. Class carries information about instantiation limit in the metamodel and solves interaction between design patterns and a class model.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)

public @interface ModelSupport {
    public String designPattern();
    public String instanceMethod();
    public int pool();
```

Figure 5. Metamodel extension

A model can be realized based on previous recommendations. For a low price – less generality – model solves all problem points identified above. Model will support to plug 3rd part components, design patterns will be instantiated in a right way. Model is executable in any time and brings immediately operational picture of modeling reality. This model is realized by reflection as a supporting mechanism for execution and debugging during program running. Reflection makes model free to use 3rd part components. Model makes instantiation of these components and other classes, calls corresponding methods defined in model and passes parameters.

3.4. Basic entity view

Graphics representation of basic model scheme suggests Fig. 6. Final list of atomic classes are available. This list represents single classes but relationships are simplified from methods to object links. In simplified model an antecedent has only one consequent and antecedent pass result process directly to consequent. Reflection realizes a passes of result and instantiation in right way with interaction to design patterns. Design pattern accuracy ensures the extended metamodel. Model input and output is defined. Every element in model has only one input and one output.

More complex metamodel presents Fig. 7., which is extension of basic model. This model is closer to reality because some entities presented in the model has more then one input. Output is limited to one because of programming language limitation. A synchronization problem occurs if method has more then one input. In this case we

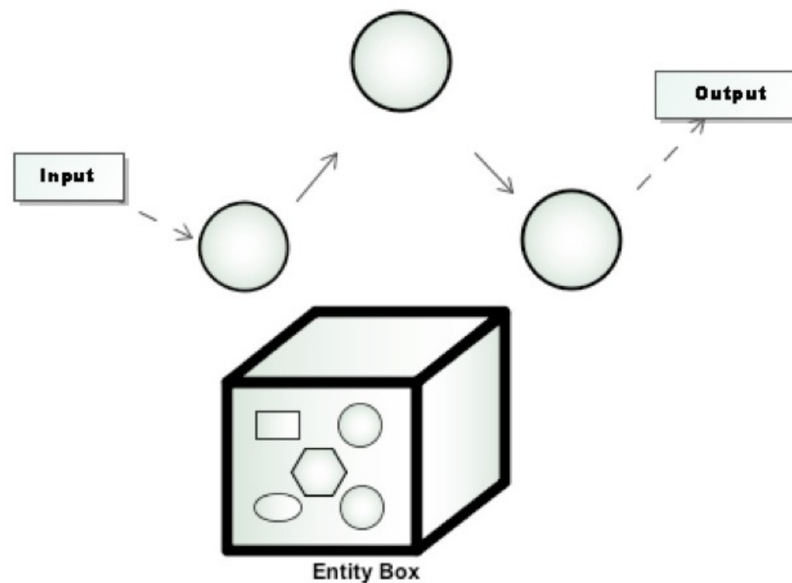

Figure 6. Basic model

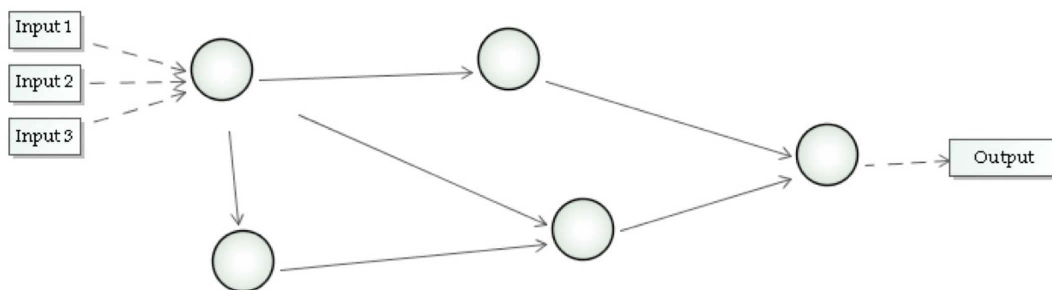

Figure 7. Extended model

must apply object-valued Petri net introduced in [1].

3.5. Class view

For an executive model representation based on reflection and annotations is more useful to create a class view. This class view shows Fig. 8., where every entity from Fig. 7. is transformed into the class. UML notation is chosen willfully because of wide using in practice. Every class contains an internal and external method. Internal methods are marked with private modifier, external with public modifier. Same approach is applied to attributes. Modeling process starts when user enters initial values and the smallest stem in simulation is one executed method. An internal state of object is changed during method execution or when the return value is generated. Returned value is passed to the next class. User can observe every object attribute and read return value after every step of execution. This feature enables reflection. User can also change interactions between objects during program running. User is able to use internal methods by changing modifiers. Internal state of the object can be edited as well. These features give user ability to create the executive model with no source code writing. This can be advantageous when result cannot be predicted but result might influent consequent components – chaining calculation. Nowadays many examples can be found. User gets possibility to create more complex structures and debug these structures after every step with no compiling. Model allows plugging some new classes during simulation.

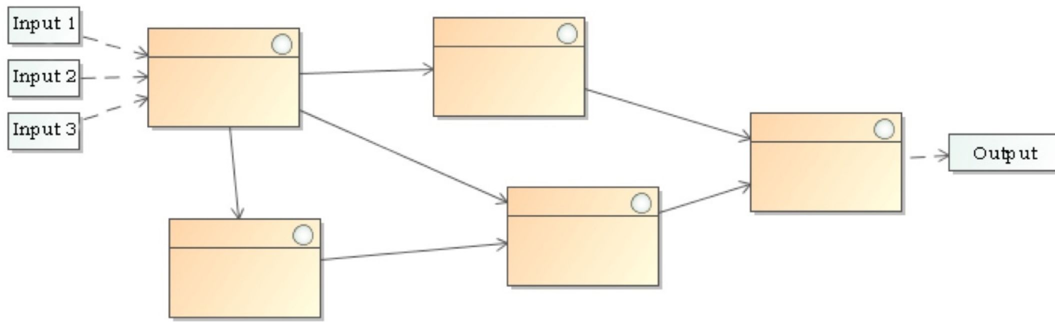

Figure 8. Extended model

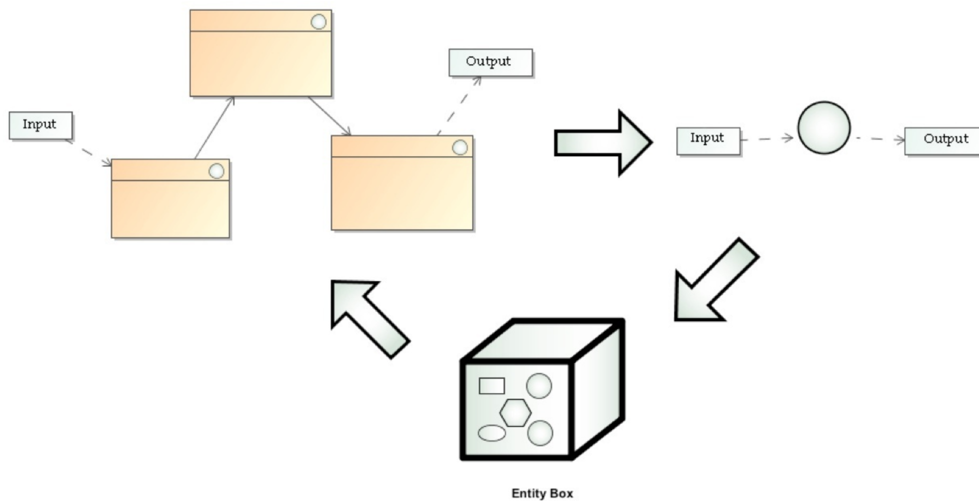

Figure 9. Level of abstraction

Metamodel, read by reflection, allows creating graphical object representation in a model. Final relationships between classes can be saved by structured XML document. XML assigns unique identifiers to classes, defines inputs and outputs and mutual return value passing.

3.6. Elevate level of abstraction

Very important model feature is ability of elevate model of abstraction. In strict metamodeling framework an instance-of operator is allowed only within layers in a same linguistic level. However if we consider ontological level we can use instance-of operator on any layer. By linking on different layers new entities arises. These entities describe [7], namely Clabject (class-object) and Powertypes. On Fig. 9. is shown a mechanism to elevate level of executive model abstraction. Model created by user consists of several classes and interactions between them. Classes are part of the entity box. This executive model is transformed to single entity after debugging and testing and carries description of significance and defines input and output point. Entity becomes a part of entity box as a single atomic element and therefore is available to future modeling of executive models. User can edit created entity and modify internal relationships or whole classes.

4. Conclusion

This paper introduces a practical proposal of new executive modeling approach introduced on [8]. First part of the paper defines problem domain and related approaches to create executive models. Following paragraph describes a reflection

application to executive model, which enables component integration. All proposals are programmed and integrated with Java programming language. Reflection can slow model processing therefore an Amdahl's law is applied to prove that there is no significant computer processing slowdown. Created model has been verified on LFLC software and brought a significant speedup during changing inferential mechanism. By implementing object-valued Petri net formalism introduced in [1] synchronization problems in complex models has been managed well. Paragraph 3.6 clarifies a possibility of elevate level of abstraction of the new executive model where the future work will continue.

Acknowledgement

This paper is supported by IGA no. 6141, Faculty of Science, University of Ostrava.

References

[1] J. Žáček and F. Huňka, "Object model synchronization based on petri net," in *Mendel 2011: 17th International Conference on Soft Computing, June 15-17, 2011*, R. Matousek, Ed. Brno: Brno University of Technology, 2011, pp. 523–527.

[2] J. Arlow and I. Neustadt, *Enterprise patterns and MDA: building better software with archetype patterns and UML*. Boston: Addison-Wesley, 2004.

[3] "ISO/IEC 10746-2:1996 information technology – open distributed processing – reference model: Foundations," 1996. [Online]. http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

[4] M. Soden, "Operational semantics for MOF metamodels." [Online]. http://www.metamodels.de/docs/tutorial_draft_v1.pdf

[5] I. R. Forman and N. Forman, *Java reflection in action*. Greenwich, Conn.; London: Manning; Pearson Education, 2005.

[6] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.

[7] C. A. González Pérez and B. Henderson-Sellers, *Metamodelling for software engineering*. Chichester, UK; Hoboken, NJ: John Wiley, 2008.

[8] J. Žáček and F. Huňka, "CEM: class executing modelling," *Procedia Computer Science*, Vol. 3, 2011, pp. 1597–1601. [Online]. http://www.sciencedirect.com/science/article/pii/S1877050911000561