ELECTRONICS AND ELECTRICAL ENGINEERING
ISSN 1392 – 1215 2011. No. 7(113)
ELEKTRONIKA IR ELEKTROTECHNIKA

SYSTEM ENGINEERING, COMPUTER TECHNOLOGY
T 120
SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Simplified Visual Modelling Approach for Executable Software Generation

M. Binkis, T. Blazauskas, E. Bareisa
Departament of Software Engineering, Kaunas University of Technology,
Studentų str. 50-101A Kaunas, Lithuania, phone: +370 604 00 386, e-mails: mikas.binkis@ktu.lt, tomas.blazauskas@ktu.lt, edas@soften.ktu.lt

crossref http://dx.doi.org/10.5755/j01.eee.113.7.613

Introduction

Current software development process is heavily based on modelling. Model-driven development promotes the role of models, allowing us to focus on the essential aspects of the system, delaying the decision of the implementation technology for a later step. In model-driven development multiple models are used, where each will address one concern, independently of the remaining issues involved in the system's development; thus allowing the separation of the final implementation technology from the business logic achieved by the system [1].

A model may specify the behaviour of a system, that is, how system interacts with external entities and changers its state over time. A behavioural model is executable if it is complete enough that the specified behaviour can be enacted or simulated by an automated execution tool.

Up until now modelling and developing were rather separated fields: the models were (and still usually are) only a guideline for developers, meaning that system architects require vast amounts of synchronization and supervision with developers in order to validate the correctness of implementation. The fact that it may be incorrect, or if there is a misinterpretation of the document, is only going to propagate itself throughout the development cycle, through high level design, implementation, and test.

An executable model is a fairly new paradigm, allowing system architects to create, validate, simulate and even implement software. This was a long awaited component for seamless transition from graphical representation of a system to a fully working programming code.

In MDA, platform-independent models (PIMs) are initially expressed in a platform-independent modelling language, such as UML. The platform-independent model is subsequently translated to a platform-specific model (PSM) by mapping the PIM to some implementation language or platform (e.g., Java) using formal rules [2]. This means that a detailed model could be easily reused, transformed and executed with vast amount of languages, even those, which don't exist yet.

As available executable modelling solutions are rather complicated and / or limited in terms of functionality, we wanted to create a new implementation independent behavioural model, capable of conveying essential concepts, but keeping the model easily understandable and readable for system architects.

In this article we will review existing visual and executable model solutions, propose a new executable model type, discuss it's implementation results and propose guidelines for further research on this topic.

Existing MDA solutions

An executable model approach has been widely discussed over past years. Some stated that models shouldn't be executable at all, as their purpose is to provide an abstract view of a system, while others have spoken in favour of executable models, claiming that this may be the future of software development.

There have been model execution tools and environments for years, even before UML. However, each tool defined its own semantics for model execution, often including a proprietary action language, and models developed in one tool could not be interchanged with or interoperate with models developed in another too. There were several initiatives, which main goal was to introduce a visual representation of system's behaviour, independent of implementation language and technology. We chose xUML and flowchart examples as the best representatives in this category.

Executable UML or xUML is a profile of UML (a subset of elements, complying to standard semantics). The main idea behind xUML is a sufficiently precise model that can be transformed into an executable form via automated or semi-automated means. xUML is composed of 4 domains:

1. Domain chart represents different aspects of a system, usually in a high level of abstraction.

2. Class diagrams represent the static side of a system. xUML profile restricts usage of certain UML elements that can be used in a xUML diagram.
3. Statechart diagrams also represent the statics and define states, transitions, events, and procedures of classes. Procedures are a composition of actions, specified in an action language.
4. Action languages are used define the dynamic side of a system: they express action semantics associated with operations and state transition diagrams in UML.

Although xUML was proposed over 10 years ago, few attempts were made to actually implement the concept. One of the main problems was lack of action language standardization – there were no widely accepted languages. Examples include OAL (Object Action Language), Shlaer-Mellor Action Language (SMALL), Action Specification Language (ASL) and others. The majority of action languages are text-based, making them less intuitive and interactive than graphical languages.

SCRALL is a graphical, platform independent object-oriented action language, used to specify relational data access, computation, data processing and low level branching decisions. It is consistent with UML 2.0 action semantics, yet the graphical notation is fairly incompatible with standard UML graphic standards. For several years SCRALL remained the only graphical action language until OMG consortium proposed and adopted a new standardized language – Foundational UML (or fUML).

The fUML standard provides a simplified subset of UML Action Semantics package (abstract syntax) for creating executable UML models [3]. The main goal of fUML was an attempt to link abstract UML models with precise semantics by ensuring sufficient subset for most object oriented and activity modelling. It was debated for some time, that creating executable UML models is difficult, because the UML primitives intended for execution (from the UML Action Semantics package) are too low level, making the process of creating reasonable sized executable UML models close to impossible [4]. That's why Action language for fUML (Alf) was introduced – a standardized action language, capable of conveying complete fUML models in textual notation. Although fUML is a comprehensive solution for creation of executable models, it is not as visual as, for example, SCRALL, making the fUML limited to only specific usage.

Am approach, connecting System Engineering and Software Engineering by applying MDE principles for supporting modelling process and generating code, has been already demonstrated by other scientists [5]. Additional means for modelling system behaviour would influence code generation possibilities.

Flowcharts have also been used as a modelling solution, mostly for learning processes. Visual programs based on flowcharts allow students to visualize how programs work and develop algorithms in a more intuitive fashion. The flow model greatly reduces syntactic complexity, allowing students to focus on solving the problem instead of finding missing semicolons [6]. Several software solutions implement flowcharts as a medium for creating executable models – examples include Iconic Programmer, SFC, Visual Logic and RAPTOR [7].

RAPTOR is the first free, open-source tool that fully supports introducing object-oriented programming, including the features of polymorphism and inheritance. RAPTOR allows students to create algorithms by combining basic graphical symbols. Students create their class hierarchy in a UML designer and then represent method bodies as flowcharts. The resulting programs can then be run in the environment, either step-by step or in continuous play mode [8]. RAPTOR is used in Carnegie Mellon University and experimental results have proven that usage of this software improved student problem solving skills while reducing the syntactical burden inherent in most programming languages [9].

Proposed modelling approach

We propose a new modelling approach, based mainly on two model types – UML class diagram to depict system's static part and a flow diagram to represent the dynamic part of a system. Flow diagram is mostly influenced by flowcharts, UML activity diagrams and SCRALL – we tried to create more visual and comprehendible model representation, while retaining completeness and executability properties.

Flow diagram currently consists of 19 elements, grouped into 4 categories:

1. Flow operations define core (essential) elements of the flow diagram. List of elements: Start, End, Input, Output. All of the elements in this category have to be directly connected by flow connection (directional connection with an arrow, pointing to the target). Fundamental flowcharts also include action element, but we divided it into several more specific action types.
2. Control statements define elements, used in branching and loops. List of elements: Decision, Decision End, Loop, Loop Break. All of the elements in this category have to be connected by flow connection.
3. Object elements and data containments define elements, associated with object oriented concepts and data storage types. List of elements: Class, Object, Collection, Property, Operation, Expression, Select. All of the elements in this category have to be connected by object connection (directional connection with a circle at the end of the connection end).
4. Object operations define flow actions, that are used in conjunction with object elements and data containments. List of elements: Create, Assign, Execute, Delete. All of the elements in this category have to be connected by both flow and object connection.

Most of the element names are deliberately chosen as abstract as possible, since it would be confusing if names would suggest certain implementation technologies. For example, "Loop" element may represent "for", "while" or other loops, but exact implementation details are hidden from the architect (although, in some cases, parameters suggest, but not force certain exact implementation).

Due to limitations of the article length, we present the most common operation examples, depicted by graphical notation with corresponding generated code (in this case – C#). Additional explanations are also included, providing extra information on less common cases.

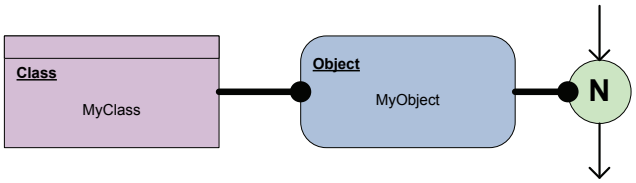

Fig. 1. Creating a new object

When creating a new object, a class name is used from the UML model and object name is entered by user. This is the most simple case of object creation, when no additional parameters are specified.

Table 1. Corresponding program text of Fig. 1

```
MyClass MyObject = new MyClass();
```

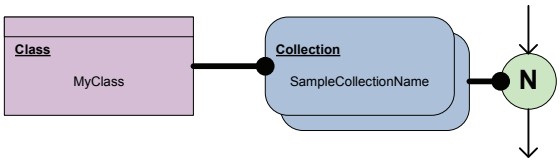

Fig. 2. Creating a new collection

Creation of a collection requires definition of a class (in this case the user specifies integer class) and a name of the collection. Currently collection depicts an array element. Both user-created and common classes (types, such as integer, string, boolean etc.) are specified by using the same element.

Table 2. Corresponding program text of Fig. 2

```
int[] SampleCollectionName;
```

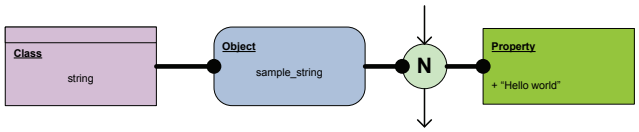

Fig. 3. Assigning a property to a created object

Properties can be assigned while creating objects. Multiple properties can be specified in the same "Property" element, each one beginning with a "plus" sign. If applicable, a single "Create" element, can be connected with multiple "Property" elements and / or multiple objects.

Table 3. Corresponding program text of Fig. 3

```
string sample_string = "Hello world";
```

When executing a method of an object, a path to a method can be represented visually. Also, a method can include multiple parameters, each one beginning with a "plus" sign. If applicable, a single "Execute" element, can be connected with multiple operations and / or multiple objects (although this practice is discouraged if the order of method execution is important).

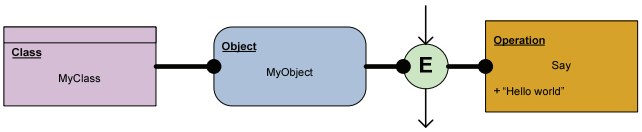

Fig. 4. Executing object's operation

Table 4. Corresponding program text of Fig. 4

```
MyClass.MyObject.Say("Hello world");
```

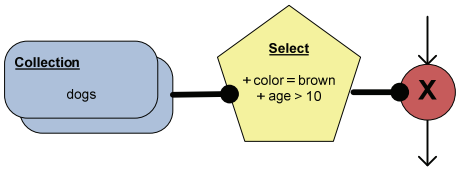

Fig. 5. Selecting objects (1)

If necessary, it is possible to select certain elements from collection and apply certain action(s) to them. In this case, objects, which properties match specified conditions, are removed from the collection. If applicable, a single "Delete" element, can be connected with multiple collections and / or multiple operations (although this practice is discouraged if the order of operation execution is important).

Table 5. Corresponding program text of Fig. 5

```
foreach (object dog in dogs)
  if ((dog.color = "brown")&&(dog.age>10))
    brown_old_dogs.Remove(dog);
```

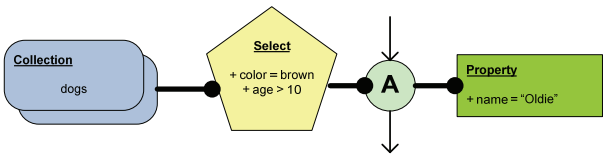

Fig. 6. Selecting objects (2)

Another example of element selection – objects with matching conditions are renamed. In this case, "Assign" element should contain only one "Property" element, since multiple properties can be defined by the single element (each beginning with a "plus" sign).

Table 6. Corresponding program text of Fig. 6

```
foreach (object dog in dogs)
  if ((dog.color = "brown")&&(dog.age>10))
    brown_old_dogs.Add(dog);
```

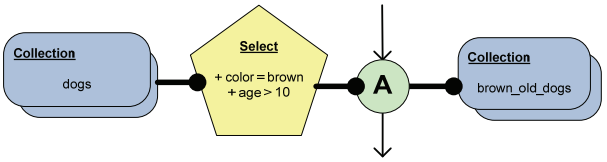

Fig. 7. Selecting objects (3)

Assigning selection results to another collection requires the usage of "Assign" element with an appropriate

collection, connected to that element. In this case "brown_old_dogs" collection must be defined earlier in the diagram.

Table 7. Corresponding program text of Fig. 7

```
foreach (object dog in dogs)
  if ((dog.color = "brown")&&(dog.age>10))
    brown_old_dogs.Add(dog);
```

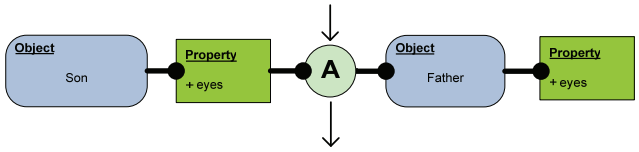

Fig. 8. Object assignment

Object or object's properties assignment involves "Assign" element, connected with corresponding objects via object connection. Similar notation is applied when assigning collections.

Table 8. Corresponding program text of Fig. 8

```
Son.eyes = Father.eyes;
```

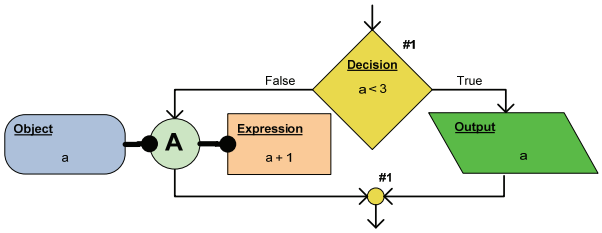

Fig. 9. Decision example (1)

Decision element has several unique properties. At first, the element has branches with named flow connectors ("True" and "False"). Secondly, it has an unique identifier (in this case we chose a number). This simple identification system significantly increases readability, when dealing with larger models, containing larger amount of decision elements. And thirdly, it has ending element ("Decision End"), with a corresponding number. Each path of a branch must be finally connected to an ending element. Currently decision element unambiguously indicates "if" control statement.

Table 9. Corresponding program text of Fig. 9

```
if (a < 3)
  a = a +1;
else
  System.Console.WriteLine(a);
```

Decision element supports multiple choices. This construction is best known as "switch" construct. If a switch value matches the value on flow connection, corresponding actions are applied. Each branch ends with loop ending element. The only connection without a value, connecting decision element with decision end element, is considered the end of decision (not a separate case).

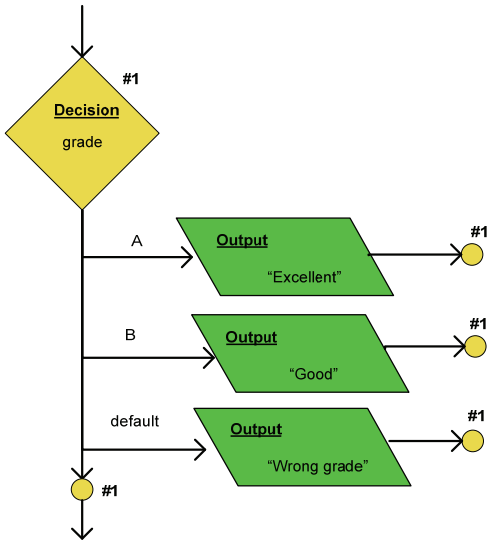

Fig. 10. Decision example (2)

Table 10. Corresponding program text of Fig. 10

```
switch(grade) {
  case "A": {
System.Console.WriteLine("Excellent"); break; }
  case "B": { System.Console.WriteLine("Good");
break; }
  default: { System.Console.WriteLine("Wrong
grade"); break; }
}
```

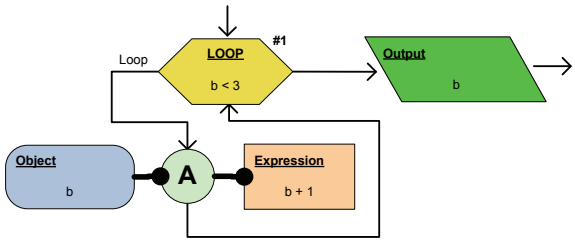

Fig. 11. Loop example (1)

"Loop" element shares some similarities with the decision element: it also has an identifier (in this case – a number) and can be ended with a "Loop End" element. Text on the flow connection ("Loop") indicates the path of a loop. The shown example would be translated into "while" control statement. In order to create a "while" statement, user must specify condition for the loop (in this case "b" is less than " 3") and the final element of a loop must be connected back to the "Loop" element.

Table 11. Corresponding program text of Fig. 11

```
if (a < 3)
  a = a +1;
else
  System.Console.WriteLine(a);
```

In this example a break of a loop is shown. When conditions are met (in this case "b" equals "1"), the loop is broken and the flow continues outside the loop. The shown example would be also translated into "while" control statement. It is important to note that loop break element

can be selected one of these values: “break”, “continue” and “restart”.

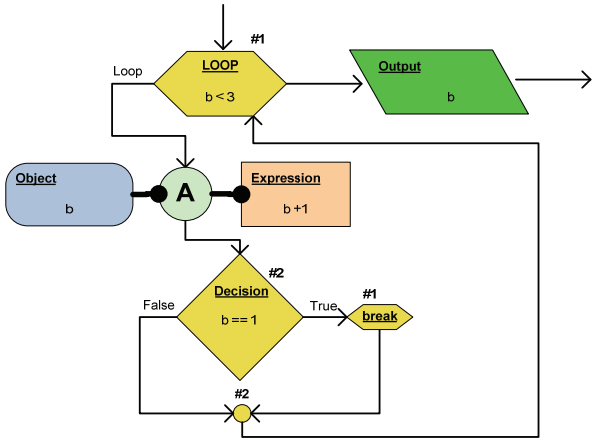

Fig. 12. Loop example (2)

Table 12. Corresponding program text of Fig. 12

```
while (b < 3) {
  b = b + 1;
  if (b == 1)
    break;
}
System.Console.WriteLine(b);
```

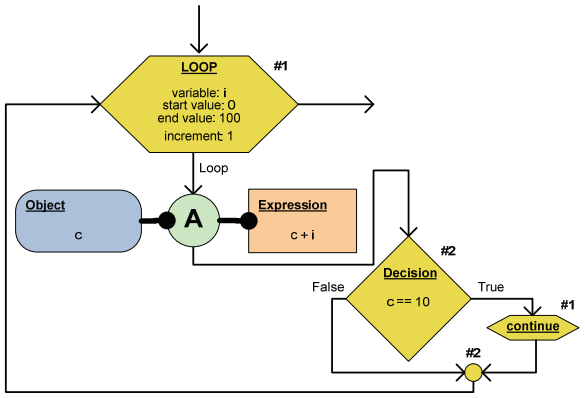

Fig. 13. Loop example (3)

Creating a loop, that would translate into “for” statement, is similar to the previous loop example, but instead of specifying a single condition, user must define several loop parameters (variable of the loop, it’s start / end values and increment of the variable).

Table 13. Corresponding program text of Fig. 13

```
for (i = 0; i < 100; i++) {
  c = c + 1;
  if (c == 10)
    continue;
}
```

A complete model of a method must include two additional elements – “Start” and “End”, connected via flow connection. Starting element should have name and would represent the name of the element. Also, it may include method name and parameters, but their visibility should be controlled by modelling tool (in case of large quantity of parameters diagram would become cumbersome). Ending element may include variables or values, being returned by a method.

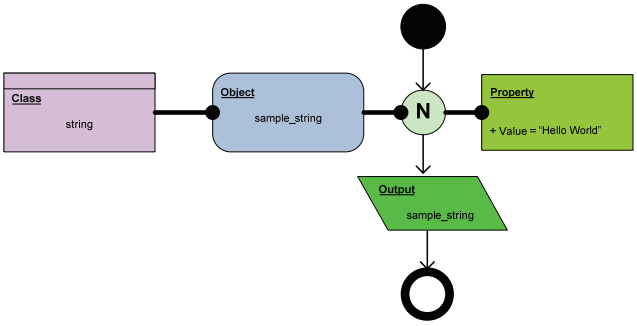

Fig. 14. Simple method

Table 14. Corresponding program text of Fig. 13

```
void HelloWorld() {
  string sample_string = “Hello world”;
  System.Console.WriteLine(sample_string);
}
```

We have implemented the presented model using Microsoft DSL Tools for Microsoft Visual Studio 2010. It was chosen because a key benefit of using a DSL is the isolation of accidental complexities typically required in the implementation phase (i.e., the solution space) such that a programmer can focus on the key abstractions of the problem space [10].

Currently it is possible to transform model into limited C#, Java and PHP code. It is important to notice that in our case only C# allows direct execution, since modelling is performed strictly in Microsoft Visual Studio environment. Our latest work includes modelling of several larger methods of automated teller machine (based on documented example, provided by Gordon College) and transforming them into fully functional programming code (C# and Java).

One of the main problems we encountered while creating this modelling approach is visibility. Although it is possible to divide model into smaller pieces, more complicated diagrams may seem unusable because of the large amount of artefacts (elements and connections between them). The solution of this problem may lie in the editor, used to create and edit models. Software, implementing our modelling approach, should be able to control the visibility of certain elements, making diagrams easier to comprehend. For example, elements, connected by object connection, could be hidden, leaving only the flow model of a method, similar to an activity diagram.

Another obstacle, limiting functionality of our proposed approach, is the lack of deployment of SDK in code generators. In order to use language native functions and libraries, code generators should include SDK of certain languages – without them the output code is limited to basic syntactic constructions and operations.

Conclusions and future research

Model driven engineering is striving to be leading methodologies of the future in software development field. Though it is debatable whether models should be left at

abstract level or include low level concepts (thus allowing execution), OMG has presented fUML and Alf as it's flagship solution, solving past obscurities of executable models. Flowcharts have also been employed as a mean for visual programming, but currently they fail to encompass the object oriented concepts. That's why we created and presented our own modelling approach, combining best qualities of today's leading solutions. Though we have encountered some problems, our modelling approach was successfully implemented and applied in practical field.

In our future research we will concentrate on refining our modelling platform and experimenting with more complicated practical applications. We are also planning to use the created tool in learning processes, mainly for teaching students basic object oriented programming skills, and present the results of our work in the upcoming article.

References

1. **Ortiz G., Bordbar B., Hernandez J.** Evaluating the Use of AOP and MDA in Web Service Development // Internet and Web Applications and Services, 2008. – P. 78–83.
2. **McNaughton M., Redford J., Schaeffer J., Szafron D.** Pattern–based AI Scripting using ScriptEase // The Sixteenth Canadian Conference on Artificial Intelligence, 2003. – P. 35–49.
3. **Lazăr C. L., Lazăr I., Pârv B., Motogna S., Czibula I. G.** Tool Support for fUML Models // Int. J. of Computers, Communications & Control, 2010. – Vol. V. – No. 5. – P. 775–782.
4. **Lazăr C. L., Lazăr I., Pârv B., Motogna S., Czibula I. G.** Using a fUML Action Language to construct UML models. // 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2009. – P. 93–101
5. **Pavalkis S., Nemuraitė L., Tarvydas P., Noreika A.** Specification of Finite Element Model of Electronic Device using Model driven Wizard–based Guidance. // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 2(98). – P. 59–62
6. **Powers K., Gross P., Cooper S., McNally M., Goldman K. J., Proulx V., Carlisle M.** Tools for teaching introductory programming: what works? // Proceedings of the 37th SIGCSE technical symposium on Computer science education (ACM), 2006. – P. 560–561.
7. **Binkis M., Blažauskas T.** Implementation of Extensible Flowcharting Software Using Microsoft Dsl Tools. // Information Technologies' 2010: proceedings of the 16th International Conference on Information and Software Technologies, 2010. – P. 211–217
8. **Carlisle M. C.** Raptor: a visual programming environment for teaching object–oriented programming. // Journal of Computing Sciences in Colleges, 2009. – Vol. 24. – Iss. 4. – P. 275–281.
9. **Carlise M. C., Wilson T. A., Humphries J. W., Hadfield S. M.** RAPTOR: introducing programming to non–majors with flowcharts // Journal of Computing Sciences in Colleges, 2004. – Vol. 19. – Iss. 4. – P. 52–60.
10. **Wu H.** Automated Generation of Testing Tools for Domain–Specific Languages. // Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ACM), 2005. – P. 436–439.

Received 2011 05 04

M. Binkis, T. Blazauskas, E. Bareisa. Simplified Visual Modelling Approach for Executable Software Generation // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 7(113). – P. 57–62.

The idea of visual executable model was one of the goals, raised by model driven engineering. Emerging software development technologies and new standards have provided the base ground for the concept to be realized in practice – SCRALL and fUML are one of the examples. Existing modelling methods are either too complex and not visual enough, or still aren't sufficient enough to depict required information for a complete executable model. We are presenting a new modelling approach, consisting of UML class diagrams and flow diagrams, capable of conveying enough information for an executable model, while retaining the model fairly visual and comprehendible, compared to the competing solutions. Ill. 14, bibl. 10, tabl. 14 (in English; abstracts in English and Lithuanian).

M. Binkis, T. Blažauskas, E. Bareiša. Supaprastintas vizualaus modeliavimo būdas vykdomosioms programoms kurti // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2011. – Nr. 7(113). – P. 57–62.

Vizualaus vykdomojo modelio idėja buvo vienas iš modeliais paremtos inžinerijos tikslų. Tobulėjančios programinės įrangos kūrimo technologijos bei nauji standartai šią koncepciją leido įgyvendinti praktiškai – SCRALL bei fUML yra vieni iš pavyzdžių. Esami modeliai yra arba per sudėtingi ir nepakankamai vizualūs, arba nepakankamai išbaigti, kad galėtų atvaizduoti vykdomajam modeliui sukurti reikalingą informaciją. Pristatomas naujas modeliavimo būdas, susidedantis iš UML klasių diagramų ir srautų diagramų, apimantis pakankamą informacijos kiekį, reikalingą vykdomajam modeliui sukurti, ir išlaikantis pakankamą modelio vizualumą ir suprantamumą, palyginti su jo konkurentais. Il. 14, bibl. 10, lent. 14 (anglų kalba; santraukos anglų ir lietuvių k.).