MATHEMATICAL AND COMPUTER MODELING

DOI 10.20535/2411-1031.2021.9.1.249812
UDC 004(942+ 4'242)

VOLODYMYR SOKOLOV,
DMYTRO SHARADKIN

PROGRAMMING MODEL BASED ON METAMORPHOSIS OF ACTIVE DYNAMIC COMPOUNDS OF OBJECTS

The paper presents the results of research on the creation of the programming model that uses structural changes in the program during execution. The urgency of the work is due to the fact that during the creation of large software systems there are problems associated with the complexity of their creation and maintenance, excessive consumption of memory and a large amount of time during execution. The base architecture is an integrated object architecture in which the program consists of active dynamic object compounds that can form dynamic connections with other objects to perform calculations. The cases when sequential transformation of compounds is necessary to reduce the complexity of the program are considered and the corresponding theoretical substantiation is given. The concept of metamorphosis of the program at runtime as a sequence of stages of changing the set of objects and the topology of their connections is introduced. The set of metamorphosis operations that affect the structure of the compound, as well as the set of operations for changing the state of compounds that determine changes in the values of the input and output connectors of objects are determined. Two kinds of metamorphosis are considered, with complete and incomplete transformation. Most attention is paid to metamorphosis with incomplete transformation when subset of the objects of the previous stage passes to the next one. Three equivalent forms of programming model have been developed: a structural model, a lifeline model of objects and connections, and an operational model. The structural model specifies the topology of the compounds of each stage without transformation operations. The lifeline model identifies the stage numbers at which objects and connections are created and destroyed, without explicitly defining the topology of the stage compounds. The operational model is specified by operations of formation of stages of metamorphosis without explicit definition of structure of compounds. Each form of the model contains state change operations that perform data input, objects activation, and output of results. The formal syntax of the structure of the program generated from the model is presented, as well as the cases of use and application in integrated objects in the form of isomeric metamorphosis and metamorphosis of the zero cycle. The obtained results allow simplifying the structure of the program and reduce the amount of code that can be generated automatically.

Keywords: program metamorphosis, active dynamic compounds of objects, programming model, software engineering.

Problem statement. When creating large software systems, there are problems associated with the complexity of their creation and maintenance (so-called “software entropy”), excessive consumption of memory, and high time spent on execution. As stated in Wirth’s empirical law, “Software is getting slower more rapidly than hardware is becoming faster”.

The complexity is affected not so much by the size of the system (number of elements and their functions) as by the number and variety of connections of elements. The main principle of complexity reduction is the hierarchical decomposition of complex systems into simpler subsystems, which in software engineering are called modules. The consumption of memory is affected by the number of elements of the system that are simultaneously in the memory. The way to reduce the consumption of memory is dynamic memory management on the principle to create elements later and destroy them early if the software architecture allows it. Reducing program execution time is achieved by reducing redundant computing and parallel execution, if, again, the program architecture allows it. Therefore, finding ways to reduce the complexity, memory usage, and execution time of programs remains a pressing issue. To address this issue, it's necessary to develop new software architectures and programming models that allow performing hierarchical decomposition, dynamic memory management, and performance management at runtime.

Imperative programming paradigms are characterized by certainty and manageability of calculations, while declarative paradigms provide a greater level of abstraction and have the potential to reduce the complexity of creating large systems. Therefore, according to the authors, the use of a multi-paradigm approach to the creation of large systems is promising, when the lower-level elements are built imperatively, and declarative paradigms are used at the upper levels, which allows building a program of larger and managed "building" blocks. For example, in the architecture of integrated objects (AIO) [1], which is the development of the authors, a special style of creating objects with elements of a functional approach to the computational model is applied and demonstrated the use of SQL-like language to describe the program with the ability to generate complete program code in C++ [2].

The disadvantages of object-oriented programming (OOP), as a kind of procedural modular programming, are the lack of mathematical basis, the opacity of inheritance mechanisms, the presence of a large number of types (classes) that are "smeared" throughout the program, the difficulty of understanding the semantics of the program. As a result, OOP does not solve the problem of complexity but contains the means of modular decomposition (the module is considered a class) and performance management at runtime. Application of model-driven architecture (MDA) using, for example, Unified Modeling Language (UML) based on OOP has promising prospects, but all attention is focused on creating a model of the domain and its mapping into action language, but the architecture of the program remains out of attention. Little attention is paid to the problem of describing and analyzing structural changes of the program during execution. For example, in UML, the issue of creating and destroying objects is only reflected in a certain way in the sequence diagram, but it does not specify the exact moment of creation and destruction of objects. Most diagrams that show the dynamics of the program actually show the dynamics of data processing (changes in the state of objects), rather than the dynamics of changes in the structure of the program.

Functional programming (FP) uses a composition of functions as a programming model, but due to the complexity of solving certain classes of problems, real FP languages deviate from the pure functional model and are supplemented by certain features of imperative languages. The composition of the functions in the FP is mostly static structures, there is no means to control the dynamic compositions, memory, and performance.

Typical topologies of compounds of integrated objects, considered in [3], allow to represent programs by schemes of compounds with static structure but do not use dynamic possibilities of compounds to the full. Therefore, a programming model is needed that will provide a representation of:

- operations of creation and destruction of objects as a mechanism of memory management;
- dynamic scheme of interaction of objects;
- the order of calculations.

Thus, the essence of the problem is the lack of models that reflect structural changes in the program during its execution, the difficulty of representing programs in static schemes, and the lack of acceptable for code generation tools of describing programs that change their structure during execution.

Analysis of recent research and publications. The analysis of recent publications and research has shown that the current direction of researching is the development of new and improvement of existing methods and tools for the high-level, close-to declarative, presentation of software models with automatic source code generation for imperative programming languages.

Thus, in [4] - [7] model-driven architecture and ways of automatic transition from the design of the system, the model of which is presented in UML, to the development of program code with minimal manual intervention are considered. This direction is supported by the development of executable UML (xUML), which influenced the creation of the Foundational UML standard (fUML). fUML is based on Action Language for Foundational UML (ALF), which is a standard action language specification by the Object Management Group. ALF specifies the executable semantics of the subject area.

One of the significant disadvantages of ALF is no support for the implementation of specific constructs, such as aggregation and composition. UML actions, as elements of activities, contain object actions and link actions (read, write, create, destroy association). Object actions include both object access operations and creation and destruction operations. In [4] a generalized analysis of xUML is given and conclusions are obtained, among which are the following:

- the most commonly used to represent models are diagrams of classes, states, and activities;
- the most popular languages for describing actions are UML actions, C ++, Java;
- areas of improvement are enhanced observability of executing models, enhanced control of model execution, and direct compilation of the model to executable files.

In [5], a design pattern-based implementation of a state machine with hierarchical, concurrent, and history states is presented, which keeps the semantics of state hierarchy and concurrency and history state as well that allows improving the quality of the generated code. In [6] a general analysis of the problems of code generation from UML-diagrams was carried out and the general conclusion was made that "there is still a huge gap in automatically transforming all UML diagrams to several source codes for use in a wide variety of platforms". In [7], several textual and graphical tools for ALF implementation are considered to supplement the executed semantics with program code texts that define the structure and behavior of the model, with automatic code generation in Java. [8] also considers the solution to supplement UML-diagrams with profile texts as a hybrid representation of the model with different notations, that reduces the effort to synchronize the different notations of the model.

Other non-UML research works also use graphical notation, but for a lower level of program representation. Thus, in [9] a general analysis of tools for visual programming languages (VPL) was conducted, and all the VPL-tools were classified as follows:

- form-based tools for building a functional user interface by dragging visual components into the form;
- diagram-based tools for building a program by connecting visual components, where the output of a component is the input of data for another component, which may represent data sets and algorithms, or simply represent graphical components;
- block-based tools for building a program by combining visual blocks that connect like a puzzle, where blocks represent programming constructs such as variable settings or loops, or interactive components such as a map;
- icon-based tools for building a program by connecting icons to represent the data flow, when icons represent services such as locating a user or saving a file, for example, allowing end users to create mobile applications.

In [10], the issue of role-based component interaction using set-oriented programming (SOP) is considered to be one of the latest collaboration paradigms. The SOP paradigm is based on two abstractions: component and set. A component is an uncomplicated entity that represents collaboration. It defines the roles of the objects involved in the cooperation. Objects that have the same role in the component (collaboration) form a set. In SOP, an application is a collection of components. One of the components is the main component, which determines the complete behavior of the program. Others must be created in order to be used. This facilitates code reuse, improves the structure and understanding of the program.

In [11] the approach to the organization of interaction of components by means of the declarative approach based on queries for manipulation of relations between attributes of various objects is presented. The rule-based write operation allows to update the status. Control is achieved through queries that select the mixing classes that are active in each object. Dynamic activation and deactivation of declarative classes of mix allows to decompose functionality into small classes that are used repeatedly. The programming style in a language is functional and reactive, and functional programs define the members of an object. Queries are a type of function that also serves as an adhesive that combines these functions to provide input. Because queries declaratively describe what they return, they leave the system to implement the method of receipt. Combining this with the organization around objects makes the language extremely suitable for complex interactive applications managed by large amounts of data from different sources. The works of the authors [1] - [3] are devoted to the consideration of various aspects of AIO, the application of functional and relational models in OOP, and the topology of ADC schemes.

Thus, the analysis of recent publications and research shows that the main direction of efforts is aimed at developing formal and graphical tools for modeling the interacting components of software systems with automatic generation of the maximum possible code in imperative programming languages. Insufficient attention is paid to the issue of modeling the dynamics of changes in the structure of interaction of system elements and their positive consequences.

The purpose of the article is reducing the complexity of software built on AIO by developing a programming model that comprehensively describes the dynamically changing structures of the programs, allowing easy source code generation.

The main material research. In the process of studying the methods of applying ADC technology to solve various practical problems, there was a problem of complexity of the representation of schemes of compounds in some cases. The analysis of such cases led to their classification into the following classes:

- cases when the scheme of program presentation has excessive complexity for its understanding and practical application;
- cases where the complexity of the compound can be obviously simplified.

The greatest complexity is caused by functions which at the calculation of a certain problem should be applied to a sequence of values of arguments, especially when it has more than one argument and set of their values are a result of the application of other functions:

$$\{y_i\} = f(\{x_i \mid x_i = f_i(z_i), i = \overline{1,n}\}). \tag{1}$$

For example, the calculation of two values

$$\{y_1, y_2\} = f_3(\{f_1(x_1), f_2(x_2)\})$$

is essentially a sequential application of the function f_3 to the results of the calculation of the other two functions:

1. $y_1 = f_3(f_1(x_1))$.
2. $y_2 = f_3(f_2(x_2))$.

The essence of the problem lies in the need to sequentially change the input of the function f_3. For this example, it is possible to use a static scheme of the compound with switching connections, which contains two switches s_1, s_2 and two-step activator *a* (Fig.1).

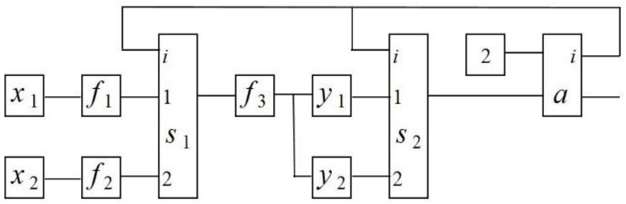


Figure 1 – Switching scheme with two-step activator

Such a scheme can be too complex if the functions have more than one argument and must be applied to more than two consecutive values. Another approach involves abandoning the static scheme and splitting the computational process into successive steps, each of which changes the structure of the compound. The first option offers an implementation with two outputs y_1, y_2 and successive changes in the structure of the compound (Fig. 2). The second option also involves a sequential change in the structure of the compound with a change in the value of one instance of the output *y* (Fig.3).

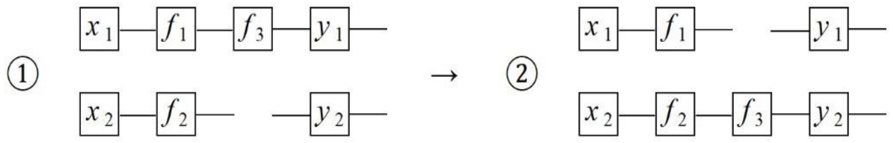


Figure 2 – Transformation of the scheme with two outputs

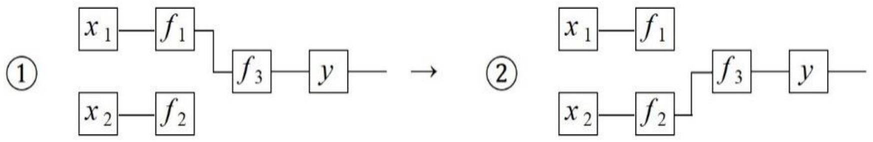


Figure 3 – Transformation of the scheme with one output

On the other hand, if consider the topology of complex compounds that do not have similar problems, we can find excessive computational complexity, if certain objects of compounds have a sufficiently high reuse rate. The basis for the sequential decomposition of a compound is the fact that objects in an adequate (calculated) state are able to store the values of input and output connectors (arguments and function values) even when the connections are broken.

Thus, any compound that implements a composition of functions $f_2(f_1(x_1))$, taking into account that

$$f_1 : x_1 \to y_1,\ f_2 : x_2 \to y_2,\ x_2 = y_1,$$

can be divided into successive stages of calculations (Fig. 4):

1. $y_1 = f_1(x_1)$.
2. $y_2 = f_2(y_1)$.

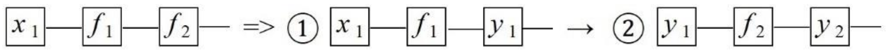


Figure 4 – Scheme of decomposition of nested functions into successive stages

For example, Figure 5 shows a scheme in which the object f_3 is activated many times, even without changes in input values. Such a scheme can also be subjected to decomposition.

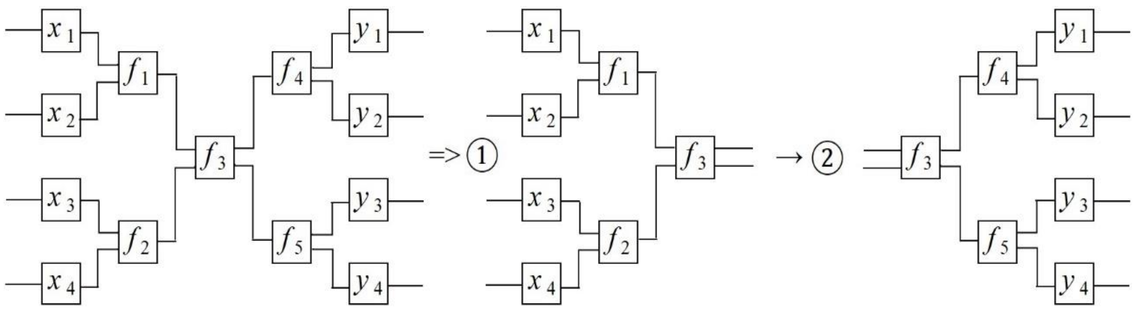


Figure 5 – Decomposition of the compound with excessive activation

In fact, such a decomposition corresponds to the partition of a large compound into modules, which must take into account the principle of “strong” links within the module and “weak” links between modules. A *nodal object* is an object that has a lot of links. Considered as a candidate to be a border to perform the decomposition of the compound to the stages. A *transit object* is an object that transfers data from one part of a compound to another between stages. Often, a nodal object becomes a transit object after decomposition (f_3 on Fig.5).

The above examples show that:

- static switching schemes are more complex, given the number of objects and connections;
- schemes with the transformation of the structure of the compound are simpler, allow dynamic changing the set of objects and connections (at each stage we can have only the necessary objects);
- schemes with excessive activation can be simplified by sequential transformation.

Metamorphosis. Changes in the program that occurs over a period of time can be considered as a *metamorphosis*. There are two classes of metamorphosis:

- *life cycle metamorphosis* (LCM) – metamorphosis that occurs during the life cycle of the program by changing the source text, the composition of the modules and the structure of their interaction;
- *run-time metamorphosis* (RTM) – metamorphosis that occurs during the execution of the program by changing the composition of program objects and their connections.

RTM is the subject of research of this work and by metamorphosis, in this work, we will understand the only metamorphosis during execution.

Consider the definitions. C*omposite* is a set of unconnected compounds. In the general case, the program in the AIO can be considered as a composite consisting of one or more compounds. *Metamorphosis* is a sequential process of changing the structure of a program, consisting of *stages*, each of which is determined by the set of objects and the topology of objects connections.
The *structural formula of metamorphosis* determines the structure of the program at each stage:

$$M = \{M_i \mid M_i =< Obj_i, Con_i >, i = \overline{1,m}, m \ge 1\}, \qquad (2)$$

where $Obj_i = \left\{ClassName_{ij} :: ObjName_{ij} \mid j = \overline{1, J_i}\right\}$ – set of objects on *i*-stage;

$Con_i = \left\{\left\langle InConnector_{ik}, OutConnector_{ik}\right\rangle \mid k = \overline{1, K_i}\right\}$ – set of connections on *i*-stage.

Another approach is to define a lifeline (LL) for each object and connection with reference to the stages of metamorphosis and to use the LL to determine the structure of the compound at each stage. The lifeline is defined by a pair of stage numbers $\langle C, D\rangle$, where *C* – the number of the stage at the beginning of which the creation occurs, *D* – the number of the stage at the end of which the destruction occurs, $C \le D$.

Then the *formula of metamorphosis on the basis of LL* can be set as follows:

$$M_L = \left\langle L_{obj}, L_{con}\right\rangle, \qquad (3)$$

where $L_{obj} = \left\{\left\langle C_k, D_k\right\rangle \mid k = \overline{1, K}\right\}$ – set of LL of the objects;

$L_{con} = \left\{\left\langle C_n, D_n\right\rangle \mid n = \overline{1, N}\right\}$ – set of LL of the connections.

Consider the definitions. *Local objects* are objects whose LL begins and ends at the same stage of metamorphosis. *Global objects* are objects whose LL begins in the first and ends in the last stage of metamorphosis. Other objects are considered *non-local*.

We will consider the following types of metamorphosis:

- *metamorphosis with complete transformation* (MCT) – when the next stage does not contain the objects of the previous stage*;*
- *metamorphosis with incomplete transformation* (MIT) – when some objects of one stage transfer to the next stage.

The program, which is a composite of compounds, in terms of metamorphosis can be represented by one of the following methods (Fig.6):

1. By including all compounds of a composite to the first stage (create all – activate all – destroy all), and it is possible to activate them both consistently, and in parallel.
2. By putting each compound of a composite in a separate stage with consecutive full transformation, and executing operations consistently for each stage (create – activate – destroy).

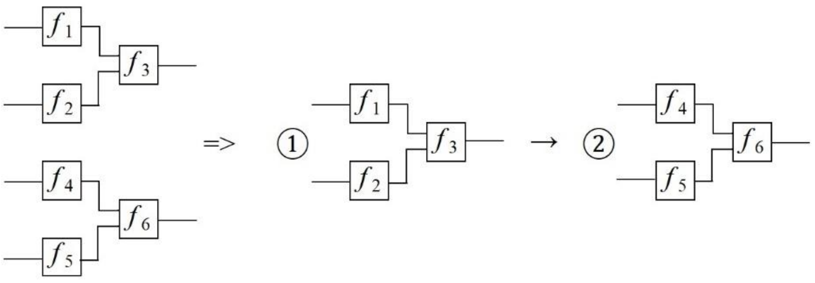


Figure 6 – Representation of the composite as MCT

Programming model. To clarify the essence and describe the transition from one stage of metamorphosis to another, consider the operations that affect the transition processes.

Metamorphosis operations include operations that change the structure of the compound, but do not change the state of objects:

- creation of an object;
- establishing a connection;
- destruction of the object;
- disconnection.

The operations of changing the state of the compound objects include the following operations:

- setting the values of the input connectors of objects;
- activation of objects and getting values of their output connectors.

For a program that contains only one stage, it is enough to create and connect all objects, set input values, activate the outputs of objects, output the required results, and then destroy all objects. But due to the fact that each individual operation of metamorphosis changes the structure of the compound, the question arises when the formation of stage begins and ends.

Let's assume that the formation of the stage begins with the first change in the structure of the compound of the previous stage and ends with the first change in the state of the objects of the compound. In essence, the beginning of the formation of a new stage can be considered a process of transition to the next stage, including case if the stage is only one. This may be a formal sign of determining the stages of metamorphosis during reverse-engineering.

When transiting from one stage to the next in the case of MIT, some objects and connections move to the next stage, and some are destroyed.

The *programming model* can be defined by three various forms.

1. *Structural model* (SM) – explicitly specifies the topology of compounds and changing the state of the objects of each stage, and taking into account (2) has the following form:

$$SM = \{< Obj_i, Con_i, SO_i > | \, i = \overline{1,m}, m \geq 1\}, \tag{4}$$

where SO_i – state change operations block of *i*-stage.

2. *Life line model* (LM) – the topology of compounds of each stage is specified implicitly by the LL and taking into account (3) has the following form:

$$LM =< L_{obj}, L_{con}, \{SO_i \, | \, i = \overline{1,m}, m \geq 1\} > . \tag{5}$$

3. *Operational model* (OM) – the topology of compounds is implicitly specified by metamorphosis operations, and changing the state of the objects are specified explicitly:

$$OM = \{< MO_i, SO_i > | \, i = \overline{1,m}, m \geq 1\}, \tag{6}$$

where MO_i – metamorphosis operations block.

All models are equivalent and each model can be derived from any other. The structural model (4) is more suitable for structural analysis, the life lines model (5) allows to control the dynamics of objects and interactions, and the operational model (6) is well suited for describing the implementation and automatic code generation of the program.

Ways of describing programming model. The representation of the model can be implemented in various ways for practical application for software development.

For designing:

– graphically in the form of a sequence of schemes of compounds of each stage of metamorphosis;
– by formulas of compounds of each stage;
– by tables of schemes of compounds of each stage.

For implementation:

– by LL tables of objects and connections;
– by using SQL-like language;
– in the form of the program text.

The transition from the design to programming can be done manually or by automatically generating the program text.

Source code generation. The generation of the program looks rather trivial: the program is a sequence of stages, each of which is represented by blocks of creation of objects, creation of connections; setting inputs, activating of outputs, output the results; destruction of connections, destruction of objects. The structure of the program in Backus-Naur Form is as follows:

Listing 1

```
Program::={Stage} {DeleteObject}
Stage::=<MetaOperations><StateOperations>
MetaOperations::={ NewObject | NewConnection | DeleteObject | DeleteConnection}
StateOperations::={[<InConnector>.set(<value>)] | <OutConnector>.get() | [OutputData]}
```

The syntax of the operations is determined by the target language and programming model.

Integration. Compounds with metamorphosis can be integrated into classes, which will allow to reuse such objects in different schemes to solve problems. In this case, the stages of metamorphosis are used in the process of activating the function of the nucleus of the object using the following options:

– *isomeric metamorphosis* (IM): when creating an integrated object all internal objects should be created too but without connections (they become global objects for all stages), at the stages of metamorphosis only the connections should be changed, and at the end of the calculations all the connections should be destroyed (Fig. 7);

– *metamorphosis of the zero cycle* (MZC): when creating an integrated object do not create internal objects at all, in the process of calculations to create objects and connections by stages (avoid global and non-local objects), and after the calculations to destroy all objects (Fig. 8).

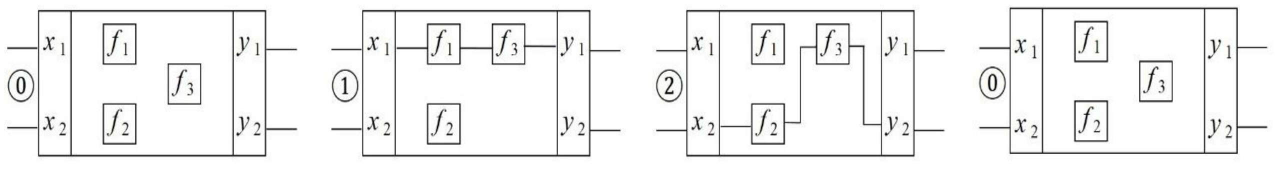


Figure 7 – Isomeric metamorphosis of an integrated object

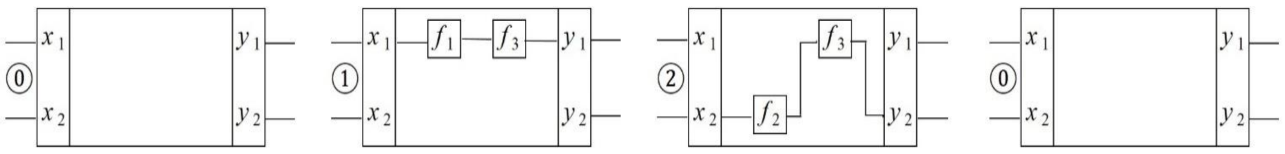


Figure 8 – Metamorphosis of the zero cycle of an integrated object

In both cases, the metamorphosis process is started only if the integrated object is not in an adequate state, or the data on the input connectors has changed. In adequate state, the values are stored in the input and output connectors. IM uses more memory, but runs faster. It's possible to put objects creation on the class constructor, change connections and activate them if necessary in the nucleus function, and destroy all objects in the class destructor. MZC, on the other hand, uses less memory and runs slower, mainly due to object creation and destruction operations. All operations rely on the nucleus function, which activates the process of metamorphosis only when necessary.

Such integrated metamorphic objects are not subject to autonomous disintegration, or require decomposition to the stages of metamorphosis of the whole compound into which they belong. Integration, on the one hand, allows the formation of a hierarchy of classes of compounds, and on the other hand allows the use of objects with linear metamorphosis in switching (branched) and iterative (cyclic) schemes.

Example. Consider the solution of a simple problem to demonstrate the application of the developed model. Given an array of numbers, we need to enter the value of the array, find the minimum value and output the elements of the array reduced by the minimum value.

This problem is difficult to solve by a combinational scheme, namely: the array requires the entry of values only once, the minimum must receive the initial value, and then change; after finding the minimum, it must be subtracted from each element during output. To solve this problem, the process must be divided into three stages:

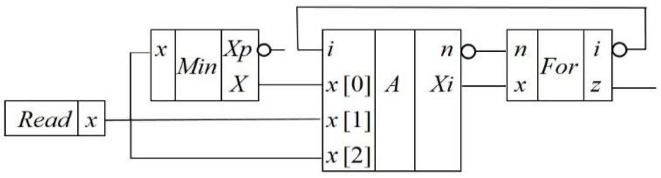
1. Input the values of the array and assign the value of zero element to minimum;
2. Find the minimum value;
3. Print the values of all elements reduced by the minimum value.

Schemes of compounds at each stage of metamorphosis can be represented graphically (Fig. 9). Analyzing the stages, we can determine the following:

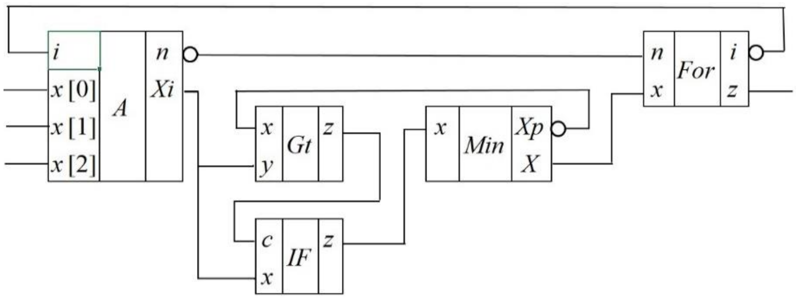
- objects 'A', 'Min', 'For' are global for all stages (reused);
- some connections of global objects are saved, and some change in the following stages, there are some local objects that are needed only at one stage;
- to calculate the compounds at each stage, it is enough to call the output 'z' of the 'For' object.

The fragment of the source code of the program in C ++ for the third stage has the following form:

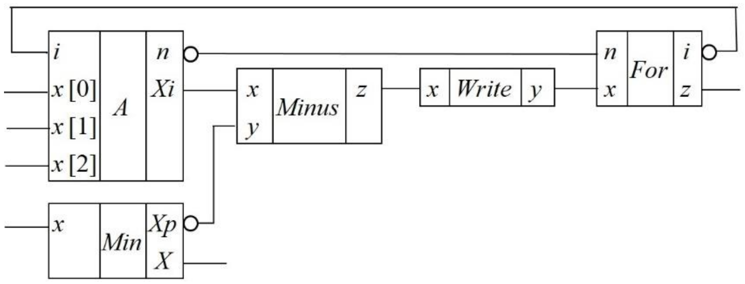
Listing 2
```
#include "Connectors.h"
…
int main(){
 …
// Deleting previous objects and connections
delete Gt;
delete IF;
Min->x.disconnect();
// Create objects of Stage3
TMinus<float>    * Minus = new TMinus<float>;
WriteCon<float> * Write = new WriteCon<float>;
// Create connections of  Stage3
Connection(Minus->x, A->Xi);
Connection(Minus->y, Min->Xp);
Connection(Write->x, Minus->z);
Connection(For->x, Write->y);
// Activate Stage3
For->z.get_value();
// Delete all objects
delete A;
delete Min;
delete For;
delete Write;
delete Minus;
return 1;
}
```



a) Stage 1



b) Stage 2



c) Stage 3

Figure 9 – Stages of metamorphosis of the compound

Use cases. The developed programming model on the basis of metamorphosis can be applied in the following cases:

- when objects need to change the input connections to solve the problem;
- there is a need to reuse the same objects at different parts of the compound;
- there is a need to simplify complex schemes of compounds;
- there is a need to increase the efficiency of calculations;
- to prevent re-entry of data and over-activation of compound with controllers;
- transferring data by transit objects between compounds.

Conclusions. The developed programming model based on metamorphosis of active dynamic compounds of objects allows reaching such positive results:

− reducing software complexity by decomposing its structure to more simple metamorphic stages of compounds of objects with sequential transformations;

− improving object lifecycle management with the ability to flexibly control memory consumption and program productivity;

− the possibility of applying the model both during software design and directly during the creation of source code manually, as well as for the formal description and analysis of the dynamics of structural change of the system;

− simplifying source code generation with the possibility of using a parallel or sequential model of calculations.

The direction of future research on this topic is the construction of conditional and iterative stages of metamorphosis.

REFERENCES

[1] V. Sokolov, "Architecture of software based on integrated objects", *Information Technology and Security*, vol. 5, no. 2, pp. 51-59, July-December 2017, doi: https://doi.org/10.20535/ 2411-1031. 2017.5.1.120559.

[2] V. Sokolov, "Application of functional and relational models in object-oriented programming", *Information Technology and Security*, vol. 5, no. 1, pp. 54-63, January-June 2017, doi: https://doi.org/ 10.20535/2411-1031.2017.5.1.120559.

[3] V. Sokolov, "Topologies of schemes of active dynamic compounds of objects", *Information Technology and Security*, vol. 7, no. 1, pp. 56-68, January-June 2019, doi: https://doi.org/ 10.20535/2411-1031.2019.7.1.184324.

[4] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of UML models: a systematic review of research and practice", *Software & Systems Modeling*, vol. 18, no. 18, pp. 2313-2360, 2019, https://doi.org/10.1007/s10270-018-0675-4.

[5] E. V. Sunitha, and S. Philip, "Automatic Code Generation From UML State Chart Diagrams", *IEEE Access*, vol. 7, pp. 8591-8608, 2019, doi: https://doi.org/10.1109/ACCESS.2018. 2890791.

[6] Maryam I. Mukhtar, and Bashir S. Galadanci, "Automatic code generation from UML diagrams: the state-of-the-art", *Science World Journal*, vol. 13, no. 4, pp. 47-60, 2018.

[7] T. Buchmann, and A. Rimer, "Unifying Modeling and Programming with ALF", *in Proc. 2nd International Conference on Advances and Trends in Software Engineering (SOFTENG 2016)*, Lisbon, Portugal, 2016, pp. 10-15.

[8] L. Addazi, F. Ciccozzi, P. Langer, and E. Posse, "Towards Seamless Hybrid Graphical–Textual Modelling for UML and Profiles", in *Proc. European Conference on Modelling Foundations and Applications*, Marburg, Germany, 2017, pp. 20-33, doi: https://doi.org/10.1007/978-3-319-61482-3_2.

[9] M. A. Kuhail, S. Farooq, R. Hammad, and M. Bahja, "Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review", *IEEE Access*, vol. 9, pp. 14181-14202, 2021, doi: https://doi.org/10.1109/ACCESS.2021.3051043.

[10] S. Masoumi, and A. Mahjur, "Collaborative component interaction", *Ingénierie des Systèmes d'Information*, vol. 24, no. 3, pp. 321-329, 2019, doi: https://doi.org/10.18280/isi.240312.

[11] Y. Seginer, T. Vosse, G. Harari, and U. Kolodny, "Query-based object-oriented programming: a declarative web of objects", in *Proc. 14th ACM SIGPLAN International Symposium on Dynamic Languages, Boston*, 2018, pp. 64-75, doi: https://doi.org/10.1145/ 3276945.3276949.

The article was received 26.02.2021.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] В. Соколов, “Архітектура програмного забезпечення на основі інтегральних об’єктів”, vol. 5, no. 2, pp. 51-59, July-December 2017, doi: https://doi.org/10.20535/2411-1031.2017.5.1.120559.

[2] В. Соколов, “Застосування функціональної та реляційної моделей в об’єктно-орієнтованому програмуванні”, *Information Technology and Security*, vol. 5, no. 1, pp. 54-63, January-June 2017, doi: https://doi.org/ 10.20535/2411-1031.2017.5.1.120559.

[3] В. Соколов, “Топології схем активних динамічних сполук об’єктів”, *Information Technology and Security*, vol. 7, no. 1, pp. 56-68, January-June 2019, doi: https://doi.org/ 10.20535/2411-1031.2019.7.1.184324.

[4] F. Ciccozzi, I. Malavolta, and B. Selic, “Execution of UML models: a systematic review of research and practice”, *Software & Systems Modeling*, vol. 18, no. 18, pp. 2313-2360, 2019, doi: https://doi.org/ 10.1007/s10270-018-0675-4.

[5] E. V. Sunitha, and S. Philip, “Automatic Code Generation From UML State Chart Diagrams”, *IEEE Access*, vol. 7, pp. 8591-8608, 2019, doi: https://doi.org/10.1109/ACCESS.2018.2890791.

[6] Maryam I. Mukhtar, and Bashir S. Galadanci, “Automatic code generation from UML diagrams: the state-of-the-art”, *Science World Journal*, vol. 13, no. 4, pp. 47-60, 2018.

[7] T. Buchmann, and A. Rimer, “Unifying Modeling and Programming with ALF”, in *Proc. 2nd International Conference on Advances and Trends in Software Engineering (SOFTENG 2016)*, Lisbon, Portugal, 2016, pp. 10-15.

[8] L. Addazi, F. Ciccozzi, P. Langer and E. Posse, “Towards Seamless Hybrid Graphical–Textual Modelling for UML and Profiles”, in *Proc. European Conference on Modelling Foundations and Applications*, Marburg, Germany, 2017, pp. 20-33, doi: https://doi.org/ 10.1007/978-3-319-61482-3_2.

[9] M. A. Kuhail, S. Farooq, R. Hammad and M. Bahja, "Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review", *IEEE Access*, vol. 9, pp. 14181-14202, 2021, doi: https://doi.org/10.1109/ACCESS.2021.3051043.

[10] S. Masoumi, and A. Mahjur, “Collaborative component interaction”, *Ingénierie des Systèmes d’Information*, vol. 24, no. 3, pp. 321-329, 2019, doi: https://doi.org/10.18280/isi.240312.

[11] Y. Seginer, T. Vosse, G. Harari, and U. Kolodny, “Query-based object-oriented programming: a declarative web of objects”, in *Proc.14th ACM SIGPLAN International Symposium on Dynamic Languages, Boston,* 2018, pp. 64-75, doi: https://doi.org/10.1145/ 3276945.3276949.

ВОЛОДИМИР СОКОЛОВ,
ДМИТРО ШАРАДКІН

МОДЕЛЬ ПРОГРАМУВАННЯ НА ОСНОВІ МЕТАМОРФОЗУ АКТИВНИХ ДИНАМІЧНИХ СПОЛУК ОБ’ЄКТІВ

Представлено результати досліджень щодо створення моделі програмування, яка використовує структурні зміни програми під час виконання. Актуальність роботи обумовлена тим, що під час створення великих програмних систем виникають проблеми, які пов’язані зі складністю їх створення та супроводження, надмірним споживанням оперативної пам’яті та великими витратами часу під час виконання. Як базову обрано архітектуру інтегральних об’єктів, за якою програма складається з активних динамічних сполук об’єктів, що можуть утворювати динамічні зв’язки з іншими об’єктами для виконання обчислень. Розглянуто випадки, коли послідовна трансформація сполук є необхідною для зменшення складності програми та надано відповідне теоретичне обґрунтування. Введено поняття метаморфозу програми часу виконання як послідовності стадій зміни складу об’єктів та топології їх зв’язків. Визначено склад операцій метаморфозу, які впливають на структуру сполуки, а також склад операцій зміни стану сполук, які визначають зміни значень вхідних та вихідних конекторів об’єктів. Розглянуто два види метаморфозу, з повним та неповним перетворенням. Найбільшу увагу звернено на метаморфоз з неповним перетворенням, коли частина об’єктів попередньої стадії переходить в наступну. Розроблено три еквівалентні форми представлення моделі програмування: структурна модель, модель ліній життя об’єктів і зав’язків та операційна модель. Структурною моделлю задається топологія сполук кожної стадії без операцій трансформації. Модель ліній життя визначає номери стадій, на яких створюються та знищуються об’єкти та зв’язки, без явного визначення топології сполук стадій. Операційна модель задається операціями формування стадій метаморфозу без явного визначення структури сполук. Кожна форма моделі містить операції зміни стану, що виконують введення вхідних даних, активацію об’єктів та виведення результатів. Надано формальний синтаксис структури програми, що генерується з моделі, а також способи застосування в інтегральних об’єктах у формі ізомерного метаморфозу та метаморфозу нульового циклу. Отримані результати дозволяють спростити структуру програми та зменшити обсяг коду, який може генеруватися автоматично.

Ключові слова: метаморфоз програми, активні динамічні сполуки об’єктів, модель програмування, інженерія програмного забезпечення.

Sokolov Volodymyr, candidate of technical sciences, associate professor, associate professor at the cybersecurity and application of information systems and technologies academic department, Institute of special communication and information protection of National technical university of Ukraine “Igor Sikorsky Kyiv polytechnic institute”, Kyiv, Ukraine, ORCID 0000-0002-5779-7167, vsokolov@i.ua.

Sharadkin Dmytro, candidate of technical sciences, associate professor, associate professor at the cybersecurity and application of information systems and technologies academic department, Institute of special communication and information protection of National technical university of Ukraine “Igor Sikorsky Kyiv polytechnic institute”, Kyiv, Ukraine, ORCID 0000-0001-6407-8040, dmsh@ukr.net.

Соколов Володимир Володимирович, кандидат технічних наук, доцент, доцент кафедри кібербезпеки і застосування інформаційних систем і технологій, Інститут спеціального зв’язку та захисту інформації національного технічного університету “Київський політехнічний інститут імені Ігоря Сікорського”, Київ, Україна.

Шарадкін Дмитро Михайлович, кандидат технічних наук, доцент, доцент кафедри кібербезпеки і застосування інформаційних систем і технологій, Інститут спеціального зв’язку та захисту інформації національного технічного університету “Київський політехнічний інститут імені Ігоря Сікорського”, Київ, Україна.