ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

UDC 004.414.23:510.637 **doi: 10.32620/reks.2024.1.01**

Serhii HOLUB, Volodymyr SALAPATOV, Vadym NEMCHENKO

Cherkasy State Technological University, Cherkasy, Ukraine

REPRESENTATION OF THE PROGRAM MODEL USING PREDICATES

The object *of research in this article is the process of modeling programs and their subsequent development.* ***The purpose*** *of this article is to develop a methodology for describing and building software models in the form of nondeterministic finite automat. To achieve this goal,* ***a task*** *was set to improve the method for describing such models using predicates based on the MODEL CHECKING technology.* ***The result*** *of this article is a method for describing and presenting program models directly according to the chosen algorithm using predicates. If the program algorithm is chosen and described correctly, the resulting model should also be correct. The model will be a non-deterministic state machine that will not require further checking, as provided by the MODEL CHECKING technology. Structurally, the model will represent a special database, the processing of which will allow turning the model into a program in any procedural programming language. When developing parallel programs that are widely used for control in aviation, land transport, military affairs, etc., two additional states of the automaton are introduced into the model, which take into account the features of such programs. Therefore, a state monitor is provided for access to shared resources and a state protocol to process parallel branches of the program. To describe the algorithm of the program, we propose to present it in the form of a connected sequence of certain actions using predicates with the use of extended temporal logic. This description covers both the logic of the program and its branches and the specific actions at each location of the program model. With the help of this methodology, a program model of a stack algorithm was developed, which is the main component for the future automated system of processing the description of program models. The program which was created according to this technology, is currently in the testing and verification stage. The sequence of processing steps of such a model is shown in the example of a floating-point constant translation program. This program is also created using this technology in the target language assembly, has been fully tested, and has shown its functionality. This description covers both the logic of the program with its branches and the specific actions at each location of the application model.* ***Conclusions****: with a correct description of the program algorithm, an adequate model of it is built, with the help of which the program itself is created in the target procedural programming language. Note that in the conditions of the rapid development of management and control automation systems in various spheres of human activity, research on the creation of reliable based on the description of their models is an urgent problem.*

Keywords: *model; predicate; temporal logic; an indeterminate finite automaton; procedural programming language.*

Introduction

Currently, the problem of proving the correctness of the program is very relevant. It is impossible to formally prove that the program was created correctly. Therefore, the technique of creating adequate models of programs based on their formal description has recently become widespread. Proving the correctness of the model at the formal level is quite possible. Therefore, the correctness of the program will depend on its implementation according to the model. Attempts to create methods of formally proving the correctness of programs [1, 2] turned out to be practically impossible, so modern methods involve the creation of a program model and its subsequent verification. MODEL CHECKING technology involves [3 - 5] the creation of a model of the program and its subsequent verification. Therefore, this technology involves the use of a special program to build a model of the program, and after building the model, it must be tested by a special program - a verifier, which establishes the correctness of logical connections between all parts of the program. This makes it difficult to create models and purchase such additional programs. After creating a program model, you can proceed directly to the creation of the program itself, and this process may be automated. Formal proof of the correctness of the program is almost impossible to perform; therefore, modern methods involve the creation of a model of the program and its subsequent verification. Creating reliable and correct programs is very important, especially in control systems for various critical processes such as aircraft management, traffic, and military affairs programs based on their models. For instance, an error in the control program caused the accident of the BOING 737 in Indonesia in 2018 and in Ethiopia in 2019. Because of an error in the control pro-

gram of some AIRBUS A350 models, the control system must be rebooted every 149 h to prevent partial or complete loss of functionality [6]. It is proposed to create these models based on their description with extended temporal logic (TL) in the form of nondeterministic finite automata (NDFA) [7]. Thus, the correctness of the program depends on its implementation according to the model. The subject of study is the improvement of the technology for developing programme models and subsequent conversion of these models into programs.

1. The current state of technologies for the development of reliable programs

The task of the research presented in this article is to improve the technology of building error-free programs, particularly parallel programs, based on their models in the form of NDFA. Such attempts were first proposed by Hoare [1] and then by Milner [2]. Hoare proposed the introduction of some axioms with the help of which to prove the correctness of programs. Such technology turned out to be quite complex and confusing and had no practical application. The same applies to Milner's proposal for his technology, which is associated with an attempt to mathematically prove the correctness of programs. This method had no practical application too. The rest of the technologies (structural programming, modular programming, object-oriented programming) allow you to improve and structure the development of programs and are in no way related to proving their correctness. The most modern technology for building correct programs, which provides for their verification and received practical application, was the MODEL CHECKING technology. The disadvantage of the MODEL CHECKING technology is the assembly of the model based on the description of its individual parts in the form of predicates. After that, the model must be verified, i.e. control of compliance of the model with its description in the form of specifications and limitations.

Therefore, to avoid these shortcomings, this article proposes performing a direct description of the entire model using predicates. At the same time, the executive part of the predicate is a list of certain actions in the form of a sequence using temporal operators, and the logical part describes the conditions for their execution. This approach can also be applied to object-oriented programming [9, 10] when creating classes. However, when describing the software model, it is necessary to be guided by the principles of structural programming. The technology of structural programming involves the selection of an appropriate mathematical model for the program and the creation of its formal algorithm. Then, in the next stage, the data are refined, their types are determined, and the algorithm is presented in an intermediate, more detailed pseudo-language. Thus, by applying the principles of structured programming and using a modified TL, it is necessary to correctly describe the model of the program being designed. Then, the model that will be created by this method corrected and will not require further verification.

The resulting model can later be used to create the desired program.

2. Research objectives and an approach to the development of reliable programs

The existing technology of program development does not allow creation without errors. The only MODEL CHECKING technology that allows you to do this involves creating program models from the description of their individual parts. At the same time, the special program forms a complete model, which must then be checked by a special verifier program. This article offers a direct description of the program model according to the chosen algorithm. It is also proposed to further develop the technology for building program models, creating their internal representation, and then transforming them into programs in the target procedural programming language. When describing the model, a special database is formed in the form of an automaton model with states and connections between them. The description should be performed using predicates in a certain format. Predicates allow you to describe the conditions for transitioning to a certain state and the actions that should be performed in this state. In the future, the model in the form of a database should be processed on all possible branches to create an internal intermediate representation of the program. Such a representation can be trans formed into a program in the desired target programming language.

3. Creation of correct program models

The main mathematical model of this technology is the Kripke structure [3], which represents the forms of the automaton model:

$$M = (S, S_0, R, AP, L),$$

where S – is the set of states of the model;

S_0 – is the set of initial states of the model;

$R = S \times S$ – is the complete relationship between S, that is, transitions from one state to another, which may be possible;

AP – is a finite set of predicates;

$L = 2^{AP}$ – is a marking function, where each state defines a set of true predicates.

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

Therefore, it is quite natural to sequentially present the description of the program model in the sequence of interconnected predicates. At the same time, the declarative part of the predicate must describe executive actions, and its logical part – the condition for their execution. The result of the program model description is a non-deterministic finite automaton.

The beginning of any algorithm of the program in the form of state should begin with the definition of all the data necessary for its operation. Next, in subsequent states that determine the execution of the program, the actions described in the state must be converted to statements of the target programming language. Completion of the description of the state is defined as a certain part of the algorithm, which in the future should end with a branch to move to other states and should be transformed into branching operators.

3.1. Internal presentation of the model and its processing

It is proposed to traverse the tree sequentially in depth, i.e., after processing the first branch of the current state and transitioning to it, the transition to the first branch of the next state is performed, and the other branches are not yet considered. Such a sequential process during the traversal of the automaton tree must be performed until we reach a state in which there has already been a transition from other states or to a final state. Then, after the processing of all subsequent branches of such states is completed, it is necessary to return to the previous state and again proceed to the processing of branches from this state. To completely bypass the tree model, all its branches must be processed. Such a sequential complete traversal of the automaton program model tree is shown in Figure 1, which shows how, after processing in the Si state, the transition to the Sj state is performed, and then to the marked or final state, from which the return to the Sj state and the processing of its next branches are performed.

Then, from this state, the return to the state Si is shown. Next, if in the S_k state all branches are processed, there is a transition to the Si state, and only after that there is a transition to the Sm state. Thus, with such sequential processing of the states of the program model tree with subsequent returns to the previous states from the marked or final states, full processing is performed. Previously, a simple database [7, 11 - 13] from two relations was proposed to represent and further process the model, as shown in Figure 2. We define the state types as start, end, protocol state, monitor state, and ordinary states, which may or may not be marked.

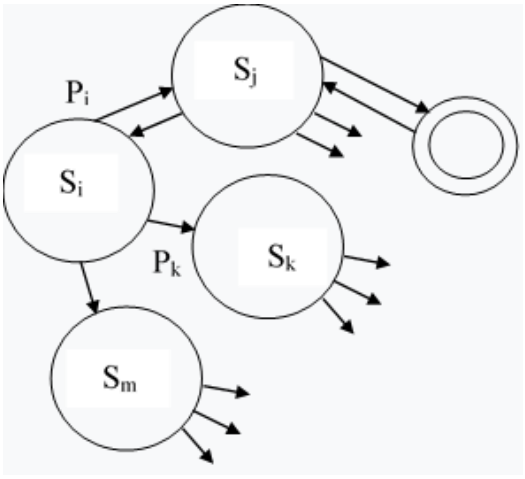

Fig. 1. Sequential processing of the automaton tree states

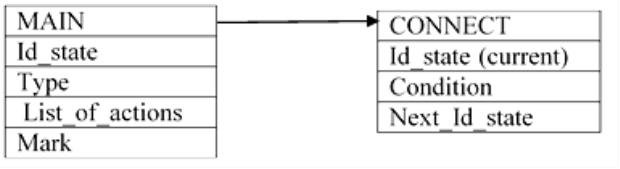

Fig. 2. Structure of the program model database

The identifier of the state Id_state determines its sequence number, and the Type attribute determines the type of vertex state. If there are multiple exits from this vertex, the Id_state attribute points to the Id_state (current) of the transition vertex. If the entrance to the peak has already taken place, the condition of such a peak should be marked to prevent readvancement in the process of bypassing the tree from other states. To completely bypass all possible branches of the automaton model, after processing the next tree branch, it is necessary to return to the previous penultimate state of the automaton and proceed to processing the next in the list of transition to another state and continue traversing the new route. When processing all possible branches of transitions from each current state, it is necessary to return to the previous state and continue traversing the tree from the next branch of the state of the previous level. That is, after completing the processing of the next branch of the automaton tree, it is necessary to perform a step back from the current vertex at the last transition. This process is an analysis into deep and is displayed as follows.

$$\begin{aligned} &S_i\,(P_j) \rightarrow S_j;\\ &S_i \leftarrow (P_j)\,S_j;\\ &S_i\,(P_k) \rightarrow S_k. \end{aligned} \qquad (1)$$

where $S_i(P_j)$ is a state of the automaton model, in which it passes on condition P_j.

Here the transition from the state S_i under the condition P_j to the vertex S_j is conditionally shown. If the bypass of the next branch at the vertex S_j is completed, then the return to the previous vertex S_i is performed according to the connection under the condition P_j. Then, the next exit from state S_i under the following condition P_k is reviewed and the transition under this condition to the state S_k is performed. The S_i state will be considered fully processed if all its outputs are processed and must be marked. The Id_state (current) attribute with respect to CONNECT is associated with the current transition state. Thus, by moving along the levels in the forward and reverse directions and moving to other branches, you can completely bypass the automaton tree. The condition for completing a complete bypass of the automaton tree model is that all states are marked. The proposed database on the one hand is a complete description of the automaton model of the program, and on the other – with the help of the Mark attribute allows you to mark processed states and perform a complete bypass of the automaton tree model. The List_of_actions attribute of the MAIN master relation is MEMO data, i.e. a string of indefinite length in which the sequence of actions in the specified state must be described. Similarly, the Condition attribute of the CONNECT relation is data of type MEMO [11 - 13] and describes the condition of transition to this state of the model. Thus, each state in the MAIN relation can be associated with several vertices in the CONNECT relation, because the NDFA model involves some transitions from one state to another.

State types include initial states, monitor states, protocol states, final states, and ordinary states. Initial states define data with their types and possible initial values. Monitor states are characterized by access to a shared resource and the presence of a semaphore variable that indicates that the resource is busy. Therefore, the monitor states independently check the occupancy of the resource, occupy it, and use it. Then, the process releases this resource. The main condition of this approach is the availability to have access of parallel branches or threads to semaphores and shared resources. That is, these data must be global. End states indicate the completion of the program and sometimes return control to the operating system. Although most control programs are cyclical, transitions to handling emergency states are possible when certain restrictions are checked.

3.2. Research on models of parallel programs

For programs with parallel threads, it is necessary to consider access to a common resource and parallel execution of several threads. The states of the protocols must wait for the completion of all parallel threads to continue the execution of the algorithm; otherwise, the state of the protocol will be in a waiting state. Because states of monitor are associated with shared resources and associated semaphores, the database of the model description is extended by another relation COMRES with the following structure, where present variable of the semaphore type – Sem (Fig. 3).

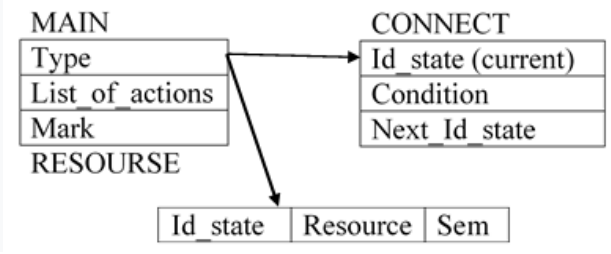

Fig. 3. Extended structure of the database

Attribute Id_state is the access key for communication with relation MAIN. At the same time, each process can capture a shared resource, and if it is free, its processing and subsequent release. In this case, the Resource and Sem are parts of the monitor integrated into the process of processing the current state of the automaton model. The Resource attribute is a reference to the shared resource, and the Sem attribute is a semaphore variable that controls whether the shared resource is occupied. Thus, states the states that need a common resource must themselves ensure the capture of the common resource and its subsequent release. The presence of such states is determined by the data type as semaphores. In the monitor states, at the beginning of the execution of certain actions, they provide for the capture of a shared resource, then perform its processing and then release the shared resource. Because the capture and release of a shared resource is controlled by a special semaphore variable, it is advisable to build the control of the semaphore variable into the corresponding states. The capture and release of a common resource can be indicated by the virtual operators LOCK(COMRES, SEM) and FREE(SEM) in the description. The COMRES parameter refers to a shared resource, and the SEM parameter refers to a semaphore variable. These links must be available to all parallel threads or branches. The monitor states are presented in Figure 4, which shows the input streams or processes S_i, S_j, S_k. Such monitor instructions are built into the operators of some modern programming languages, and access to RESOURCE and SEM-type data is implemented through the S_{mon} state.

States in which a shared resource is accessed must be surrounded by the resource capture

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

LOCK(RESOURCE, SEM) and resource release FREE(SEM) operators specified above. The generalized structure of such a state has the following form

```
COMRES  DW  <DATA>
SEM     DB      0
                .
                .
        LOCK(RESOURCE, SEM)
                ; resource processing
                FREE(SEM)
```

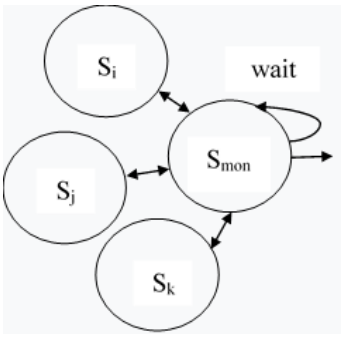

Fig. 4. State-monitor

The final states complete the execution of the algorithm and provide for the transfer of control to the operating system. The initial states of the model have an unconditional start (true) of the algorithm. To define a state type, the corresponding reference to the shared resource must be defined specially. A semaphore variable must be attached to it. In the process of traversing all branches of the tree of the model of NGFA you need to convert the List of actions attributes of the MAIN and Condition attribute of the CONNECT relation into a sequence of operators of the procedural target programming language.

Protocol states wait for parallel branches to complete before performing proceeding actions. In the case of protocol states, processing is performed after the execution of parallel branches of the algorithm model is completed before continuing with certain actions; otherwise, this state will be in a waiting state, for which the wait keyword is used in Figure 5, which shows the input parallel streams S_i, S_j, S_k.

In the process of traversal, the mathematical formulas of the description of actions for each state, as well as the description of the logical conditions of transitions to other states are transformed into a sequence of operators in the selected procedural programming language. The condition for exiting the protocol state is the completion of actions in all parallel threads. This transformation is a process of translation from the model description language into the target procedural programming language.

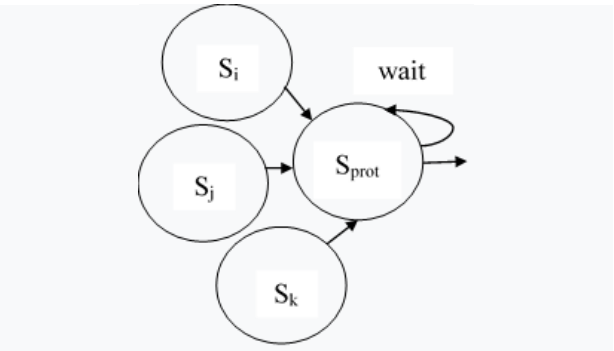

Fig. 5. State-protocol

The traversal of the model tree should be performed from the initial state (set of states) and move along the branches of the tree graph, checking the appropriate conditions of transition to another state. When the automaton moves from any state to several other states, it is necessary to sequentially process all transitions to these states and move to the next state, starting with the transition that is the first in the list of transitions. To prevent re-entry into the state that has already been processed, and to stop traversing the automaton tree on its current branch, it is necessary to mark this state in the Mark attribute. If when checking the state of the transition it turns out that it is marked, the processing of wood on this branch is stopped and the return to the previous state is performed. The second condition for stopping the processing of the next branch of the automaton tree is the transition to the final state of the automaton in its presence. The fact is that cyclic programs of the final state may not be except for emergencies, which should be provided in the description of the program model.

3.3. Results

Thus, when processing the description of the software model, the model itself is first created in the form of NDFA, which is presented in the form of a special database. In the future, the model should be converted into a program in target program language. Transitions in each state in any procedural programming language must be implemented by branch operators. The movement in the process of traversing the automaton tree is consistent with depth, and for a possible return to a certain state, the return address must be saved. A convenient mechanism for this is the stack. In this case, when successively moving to the depth of the return address, it will be consistently stored in the stack. At the end of the processing of the next branch of the automaton model of the program is the return to the previous state and the processing of the next branch. Formulas (1) illustrate the return from the state that completes the processing of any branch of the automaton model to the

previous state. Assume that this state has several transitions to other states under conditions (P_1, P_2,… P_n), then the transition from this state to the previous level state is possible if all its transitions under conditions (P_1, P_2,… P_n) are processed. In this case, such a state in relation to MAIN is marked in the Mark attribute, and in the case of processing tree on other branches, the observed state will indicate the completion of processing of the current branch. In other words, if all transitions from the current state are processed, this transition becomes marked.

$$\begin{gathered} S_i \leftarrow (P_1)\, S_j; \\ S_i \leftarrow (P_2)\, S_k; \\ \text{……..} \\ S_i \leftarrow (P_n)\, S_l; \\ \text{Mark}(S_i) = \text{true}. \end{gathered} \qquad (2)$$

$\text{Mark}(S_i)$ is an attribute of the database for state. Formulas (2) illustrate the conditions for establishing the Mark attribute in the Si state as marked, when all branches (S_j, S_k,… S_l) of this state in the states (S_j, S_k,… S_l) under the conditions respectively (P_1, P_2,… P_n) are already processed. At the end of the current branch model processing, the return address of the previous state to which you will need to return remains in the stack. Thus, sequential advancement into deep with subsequent returns provides complete processing of the program's automaton model. The implementation of specific actions in each state of the automaton is provided by translating the description in the attribute List_of_actions of the MAIN relationship. Transitions to other states are provided by translating the description in all CONNECT attributes of the CONNECT relationship related to the current state.

The description in the attributes List_of_actions and Condition is performed in terms of modified temporal logic (TL) [7] for all types of states and can be converted into a sequence of operators of the target procedural programming language. Branching from each current state into other states, as already mentioned, should be performed as a reference to parts of the algorithm model. If all branches from the current state are processed, the return to the previous state should be performed as a return from the subroutine. In this case, the return address will be stored in the stack each time, as it is implemented in all programming languages. Therefore, sequential advancement in the processing of the automaton program tree will ensure correct return to all previous states according to the description model. Thus, lower-level state processing routines are nested upper-level routines.

Earlier in [7], it was shown how monitor-states and protocol-states can be used to describe parallel program models, which greatly simplifies model verification, as suggested in MODEL CHECKING technology and previous work.

The technology for developing an automaton tree model of the program based on its description and subsequent processing to convert the model into a program is presented in Fig. 6.

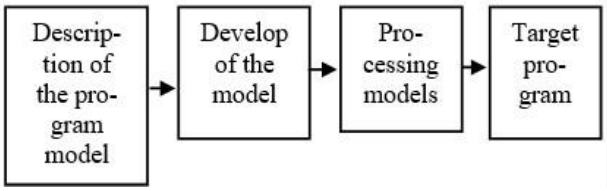

Fig. 6. Technology of creating a program by building an automaton model

The description of the program model must be performed in terms of modified temporary logic. Because of this description, a program model is created in the form of an indeterminate finite automaton, which is presented as a special database. The structure of database relationships allows you to perform complete processing of the model tree and directly create a program in the desired target procedural programming language. The description of the program model must be performed in terms of the modified TL. Because of this description, a program model is created in the form of NDFA, which is presented as a special database. The structure of database relationships allows you to perform a complete processing of the model tree of the program and go to the application of the target programming language.

This technology can also be used to describe difficult classes using object-oriented programming. Processing of List_of_actions and Condition attributes in MAIN and CONNECT relations is a translation process to which it is appropriate to apply the stack algorithm [7, 14], which involves processing the description depending on the priority of description actions. This technology can also be used to describe complex classes using object-oriented programming. Processing of List_of_actions and Condition attributes in MAIN and CONNECT relations is a translation process to which it is appropriate to apply the stack algorithm [7, 14, 15], which involves processing the description depending on the priority of description actions. o classes when using object-oriented programming. Processing of List_of_actions and Condition attributes in MAIN and CONNECT relations is a translation process to which it is appropriate to apply the stack algorithm [7, 14], which involves processing the description depending on the priority of description actions. As noted, in the development of parallel programs, special states are used. It is a state-monitor that provides access to shared re-

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

sources and a state-protocol that allows parallel threads of the algorithm to be executed.

Currently, in Cherkassy State Technological University is working to create software to describe and process program models for this technology. In particular, a stack algorithm for processing the description of actions for the List_of_actions and Condition attributes in the MAIN and CONNECT relations was developed and implemented.

4. Example

Consider an example of the translation of a constant of the type FLOAT POINT. To begin with, we will describe in detail the means of describing the model. As mentioned, we use a predicate apparatus for this. In fact, the predicate itself consists of a logical part, which we surround with curly brackets, and an executive part, which we surround with square brackets. Inside the executive part of the predicates are allowed internal predicates, which must be surrounded by parentheses. Among the operators in the conditional part, the relation operators (>, <, =, <>,> =, <=) are used to organize transitions to other states of the automatic model. Arithmetic operators (+, -, *, /) are used in expressions as elements of operators. Expressions can be present both in the operators of the conditional part and in the operators of the executive part. This example uses the virtual functions fild and cstod, the first converting an INTEGER number to FLOAT POINT and the second converting a symbol of the digit to its value. Otherwise, the keyword is also used as a condition that is the opposite of the previous condition. In addition, in case of an error, its code is generated for the subsequent output of the error type. The temporal operators U(ntil) and (ne)X(t) are also used in the operators of the executive part, which ensure the cyclic execution of the sequence of actions and their sequential execution. In addition, go to <label> transition operators are used to organize transitions to other states. Both the conditional parts of predicates and their executive parts can be marked with labels. The label should end with a colon (:). The consistent use of U operators ensures the implementation of the choice under several conditions. It is also permissible to use both standard and special functions. The latter must be described separately and end with a ret return operator. So, having chosen first the general model of the algorithm, and going step by step to the final model, we will try to describe it with the means described above.

Here is an example of the description of the constant FLOAT POINT.

1. {true}[buff; i=1,sum=0,count=0,const=0,dig,
   coder: intger; fsign=0, fneg=0, fexp=0, fdot=0:byte];
2. m4: {buff(i) = ’+’} [fsign=1, goto m1]
   U {buff(1) = ’-’} [fsign=1, fneg=1, goto m1] U
   {buff(1) =’.’} [({fexp=1} [coder=cod1, goto error]
   U {otherwise}[fdot=1, goto m1]]
   U (m2: {buff(i)=cyf}[dig=cstod(buff(i)),
   sum=sum*10+dig, {sum>smax}[coder=cod2,
   goto eror]] |
   (m3: {fdot=1} [goto m1]) U {otherwise} [goto m1])
   U ({buff(i) = (‘E’ | ‘e’)} ({fexp=1} [coder=cod3,
   goto error] U ({otherwise}[fexp=1] |
   {true}[const=fild(sum)] | {fneg=1} [const= - const,
   fneg=0] | {true} [call fin, i++, {i>=max} [stop]]
   U {oterwise}[fsign=0, fdot=0, sum=0, goto m1]);
3. m1: {true} [i++ {i>=max} [call fin, stop]
   U {othewise}[{fexp=1} [{sum=0} [stop]
   U ({sum>0} [const=const*10, sum=sum-1] |
   {true}[stop]) U ({sum<0} [const=const/10,
   sum=sum-1]) X{true}[stop] U ({fdot=1} [goto m2]
   U {otherwise} [goto m3]];
4. error: {true} [goto m1]

The first line of the description presents the initial data: buff – a buffer where the FLOAT POINT constant should be placed in symbolic form, where i is an indicator of the position of the constant symbol in the buffer, sum – the initial value of the integer constant, count – a counter of fractional digits, const – initial the value of the converted constant in the format EXTENDED, fsign, fexp, fneg, fdot = 0 – flags of the sign, exponential part and fractional point. The variable coder is designed to place the code of a possible error, cod1 – when the dot appears again, cod2 – if many digits are used to define the FLOAT POINT constant, cod3 – when the symbol of the exponential part appears again.

The second line describes the actions required to recognize the sign (state 1), fractional point (state 2), decimal digit (state 3), exponential part (state 4), and error (state 7). State 5 detects an overflow of digits. The third line describes the transition to the analysis of the next symbol of the constant with the transition to the analysis of the next symbol (state 1), and in the case of processing the exponential part – to state 4. The fourth line describes the conversion of the constant from the INTEGER format to the FLOAT POINT format (state 8). All actions in the above states are represented in the description, and labels from the description are given in front of the corresponding states for the transfer of control. The automaton model for processing program of the FLOAT POINT constant is shown in Fig. 7.

This program is a part of all compilers. Because the application model fully corresponds to its description, the next step of checking, as required by the MODEL CHECKING technology, is not required.

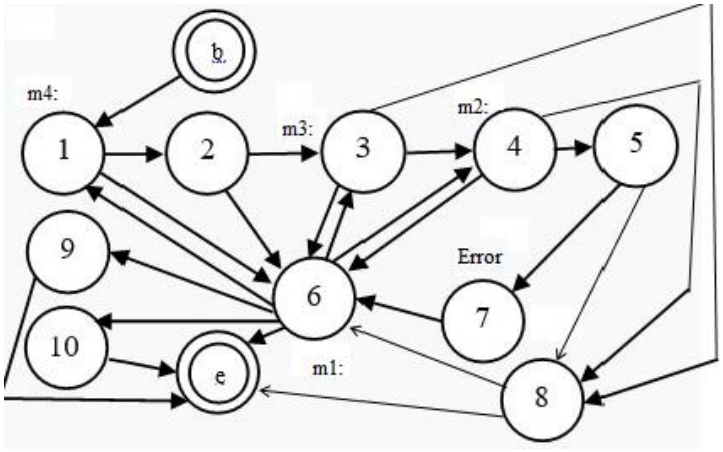

Fig. 7. Automated model for processing program of the FLOAT POINT constant

The program implementation of this example in accordance with the obtained automaton model as part of the compiler must be implemented in the ASSEMBLER language. Such implementation was carried out at the Department of Software of Automated Systems of the Cherkasy State Technological University. The program fully demonstrated its error-free operation. This technology has also been used to create parallel programs using shared resources and parallel thread,

The second line describes the actions for recognizing the sign (state 1), fractional dot (state 2), decimal digit (state 3), exponential part (state 4), and error (state 7). State 5 reveals an excess of the number of digits. The third line describes the transition to the analysis of the next symbol of the constant with the transition to the analysis of the next symbol (state 1), and in the case of exponential part processing – to state 4. The fourth line describes the conversion of the constant from INTEGER format to FLOAT POINT format (state 9). All actions in the above states are presented in the description, and the labels from the description are given before the corresponding states. This program must be part of the work of all translators and interpreters. Because the model of the program fully corresponds to its description, the next step of verification, as required by MODEL CHECKING technology is not required.

The advantages of this technology are that the creation of the program algorithm model is performed directly in its description. Therefore, with a correct description of the model, the model itself will also be correct, and unlike Model Checking technology, it does not require further verification. Note that before the exact description of the model is made, several intermediate steps must be performed. That is, first, you should create a general model, then gradually refine the model, adjust and finally form an accurate model. In the past, this procedure was performed by a group of problem solvers – algorithmists, and then the task was transferred to programmers. The proposed technology for creating models and their subsequent processing will significantly speed up the process of developing reliable programs.

The technology considered for creating program models by describing them based on the modified TL, with the correct description of the program model, allows you to obtain a suitable program model in the form of a NDFA and avoid the verification step. According to such a model, the process of transition from a model to one's own programming in the target language becomes transparent and understandable. The process of transforming the program model can be implemented on the basis of the stack algorithm [7, 14, 15]. Figure 8 explains how an application model description is transformed into a database model. Only the presentation of the first three states of the program model is shown, indicating their numbers according to the model graph. Transition from one state to another is performed through a binding relation.

The main relation of each state in the executive part of the predicate in the field List_of_actions (type MEMO) contains a list of actions that should be performed in this state according to the description of the model. The binding relation is provided by the section of the conditional part of the Condition predicate (of the MEMO type). The complete model of the program in the form of a database can be presented in a similar manner. The actions in the List_of_actions and Condition sections must be converted to statements in the target procedural programming language. In the case of automation of the process of conversion of the mentioned areas, that is, their translation, the process of describing the program model directly into the program will not require a verification step, as provided by the MODEL CHECKING technology. For this purpose, the stack algorithm can be applied, when a pair of lexemes is revealed during the lexical analysis – a data lexeme and an action lexeme [7]. Action lexemes are assigned priorities according to which the conversion process is

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

performed. Simultaneously, additional temporal logic operators will have a lower priority than the rest of the operators, which will allow the final conversion of the description into the operators of the target programming language. The use of parentheses, which are given the highest priority, allows you to change priorities during processing. This is achieved because the description of the algorithm model of the program is created consistently exactly as it was done at one time by algorithmists and passed the project of the program model to programmers. At the same time, the final verification of the program will consist only of the agreement of data types regarding their compatibility in the program.

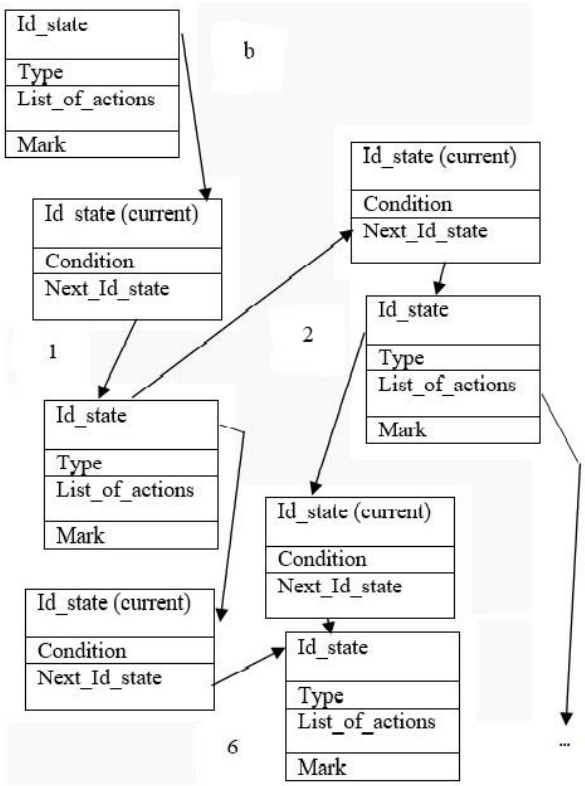

Fig. 8. Part of database of program model

5. Discussion

The proposed technology for developing program models allows the creation of such a model when describing it in the form of a special database. This model can then be turned into a program in the chosen procedural programming language, which is a new step towards the automation of program development. Means for developing parallel programs in the form of status monitors and status protocols have been created.

Compared with the most modern MODEL CHECKING technology, this technology does not require special programs for creating a model and its subsequent verification. This allows you to create correct programs if the model is correctly described. This is precisely the scientific novelty of this work. From a practical point of view, the proposed modeling and software development technology will accelerate the development of reliable programs. If this technology is put into industrial use, it will significantly speed up various computer developments. From the point of view of development prospects, they consist, first of all, in the development of software support for the proposed technology. Further development of this technology is the development of a translation program from the description language to the operators of the programming language. In addition, when using the technology for different target programming languages, it is advisable to develop the transformation of the model into an intermediate form from which the transition to the target programming language is possible. To this should be added the development of the subroutine library. All of this together opens wide prospects for the further development of this technology.

Conclusions

The proposed technology for developing models of the programs allows you to create such models when describing them in the form of a special database. This model can then be turned into a program in the chosen procedural programming language, which is a new step towards the automation of program development. Means for developing parallel programs in the form of status monitors and status protocols have been created.

Compared with the most modern MODEL CHECKING technology, this technology does not require special programs for creating a model and its subsequent verification. This allows you to create correct programs if the model is correctly described. This is precisely the scientific novelty of this study. From a practical point of view, the proposed modeling and software development technology will accelerate the development of reliable programs. If this technology is put into industrial use, it will significantly speed up various computer developments. From the point of view of development prospects, they consist, first of all, in the development of software support for the proposed technology. Further development of this technology is the development of a translation program from the description language to the operators of the programming language. In addition, when using the technology for different target programming languages, it is advisable to develop the transformation of the model into an intermediate form from which the transition to the target programming language is possible. To this should be added the development of the developed subroutine library. All of this together opens wide prospects for the further development of this technology.

The main contribution of this research is the further development of the idea of formal description and creation of models of programs, followed by the creation of reliable error-free programs. Models are described using predicates, resulting in an automaton model in the form of an NDFA in a set of states with connections. For parallel programs, additional states of program models were proposed, which correctly synchronize the execution of individual threads.

Author Contributions: a general method of presenting a model and its processing in the form of a database – **Volodymyr Salapatov**; definition of the monitor state – **Serhii Holub**; definition of the protocol state – **Vadym Nemchenko**.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

This research was conducted without financial support.

Data availability

This work has associated data in the data repository.

Use of Artificial Intelligence

The authors confirm that they did not use artificial intelligence methods while creating the presented work.

All the authors have read and agreed to the published version of this manuscript.

References

1. Hoare, C. A. R. *Communicating sequential processes*. Prentice Hall International Publ., 2022. 260 p. Available at: http://www.usingcsp.com/cspbook.pdf. (accessed 02.05.2023).

2. Milner, R. *A Calculus of Communicating Systems*. *Book series: Lecture Notes in Computer Science*. Springer Berlin, Heidelberg, 1980, vol. 92, 174 p. DOI: 10.1007/3-540-10235-3. Available at: http://www.lfcs.inf.ed.ac.uk/reports/86/ECS-LFCS-86-7/ECS-LFCS-86-7.pdf. (accessed 02.05.2023).

3. Clarke, E. M., Gramberg, O., Kroening, D., Peled, D., & Veith, H. *Model Checking*. Second edition. MIT Press Publ., 2018. 424 p. ISBN 0262349450. Available at: https://books.google.com.ua/books?id=qJl8DwAAQBAJ. (accessed 02.05.2023).

4. Zhang, Y. Ji, P.-F., Zhu, P.-W., Peng, P., Li, H.-W., & Jiang, J.-H. Parallel Software-Based Self-Testing with Bounded Model Checking for Kilo-Core Networks-on-Chip. *Journal of Computer Science and Technology*, 2014, vol. 38, pp. 405-421. DOI: 10.1007/s11390-022-2553-3.

5. Grobelna, I., Grobelny, M., & Adamski, M. Model checking of UML activity diagrams in logic controllers design. *Advances in Intelligent Systems and Computing*, 2014, vol. 286, pp. 233–242. DOI: 10.1007/978-3-319-07013-1_22.

6. *Oshibka v PO Airbus A350 vynuzhdayet perezagruzhat' sistemy samoletov kazhdyye 149 chasov* [Airbus A350 software bug forces aircraft systems to reboot every 149 hours]. Available at: https://internetua.com/oshibka-v-po-airbus-a350-vynujdaet-perezagrujat-sistemy-samoletov-kajdye-149-csasov. (accessed 02.05.2023). (in Russian).

7. Salapatov, V. I. Poryadok opysu i obrobky hrafa avtomatnoyi modeli [Order of the description and processing of the program automaton model graph]. *Matematychni mashyny i systemy – Mathematical Machines and Systems*, 2021, no 3, pp. 121-125. DOI: 10.34121/1028-9763-2021-3-121-125. (in Ukrainian).

8. Rumbaugh, J., Jakobson, I., & Booch, G. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, 1999, 568 p. ISBN 0-201-30998-X. Available at: https://idsi.md/files/file/referinte_utile_studenti/The%20Unified%20Modeling%20Language%20Reference%20Manual.pdf. (accessed 02.05.2023).

9. Lafore, R. *Ob"yektno-oriyentirovannoye programmirovaniye* v S++ [Object-oriented programming in C++]. SPb, Piter Publ., 2004. 928 p. (in Russian).

10. Eliens, A. *Printsipy Ob"yektno-oriyentirovannoy razrabotki programm*. 2-ye izdaniye [Principles of Object-Oriented Software Development Second Edition]. Sankt-Peterburg, Vil'yams Publ., 2002. 496 p. ISBN 5-8459-0233-9. (in Russian).

11. Omel′chenko, L. N., & Shevyakova, D. A. *Samouchitel' Visual FoxPro 9.0* [Visual FoxPro 9.0 tutorial]. Sankt-Peterburg, BKHV-Peterburg Publ., 2005. 608 p. (in Russian).

12. Pasichnyk, V. V., & Reznychenko, V. A. *Orhanizatsiya baz danykh ta znan'* [Organization of databases and knowledge]. Kyiv, BHV Publ., 2006. 384 p. Available at: https://www.twirpx.com/file/1174516/. (accessed 02.05.2023). (in Ukrainian).

13. Garsia-Molina, H., Ullman, J. D., & Widom, J. *Database System Implementation. United States Ed Edition*, Prentice Hall, 1999. 653 p. ISBN-13: 978-0130402646.

14. Hopcroft, D. E., Motwani, R., & Ullman, J. D. *Introduction to Automata Theory, Languages and Computation. 3rd Edition.* Pearson Education Publ., 2006. 535 p. Available at: https://e.famnit.upr.si/pluginfile.php/636821/mod_page/content/8/Automata.pdf. (accessed 02.05.2023).

15. Konvers′kyy, A. Ye. *Lohika (tradytsiyna i suchasna). Pidruchnyk dlya studentiv vyshchykh navchal'nykh zakladiv* [Logic (traditional and modern). The textbook for university students]. Kyiv, Tsentr uchbovoyi literatury Publ., 2008. 536 p. ISBN 978-966-364-735-7. (in Ukrainian).

ISSN 1814-4225 (print)
ISSN 2663-2012 (online)

Received 18.06.2023, Accepted 20.02.2024

ПОДАННЯ МОДЕЛІ ПРОГРАМИ ЗА ДОПОМОГОЮ ПРЕДИКАТІВ

Сергій Голуб, Володимир Салапатов,
Вадим Немченко

Запропоновано розв'язок вирішення проблеми підвищення надійності та позбавлення від помилок комп'ютерних програм за рахунок створення їх адекватних моделей та побудові на їх основі, власне, програм. Отримані результати дозволяють позбавитись можливих помилок при побудові різних програм, зокрема критичних програм управління в авіації, наземному транспорті, військовій справі тощо. Об'єктом дослідження є процес побудови моделей програм за допомогою модифікованої темпоральної логіки. Метою даної роботи є розробка методу побудови адекватних моделей програм, за допомогою яких можна створювати самі програми на цільовій процедурній мові програмування. На відміну від існуючої технології MODEL CHECKING цей метод не потребує кроку верифікації моделі. Це дозволяє спростити процес побудови моделі програми і самої програми, усунути помилки у програмах та підвищити їх надійність. Цей новий крок у напрямку автоматизації процесів розробки комп'ютерних програм підвищеної надійності, що якраз і складає наукову новизну цієї роботи. Опис моделі програми охоплює як логіку програми з її розгалуженнями, так і конкретні дії у кожному місці моделі програми. У результаті такого опису створюється модель програми у вигляді невизначеного скінченного автомату, яка, у разі коректного його опису, дозволяє розробляти коректні моделі і у подальшому програми. Формально модель представлена за допомогою спеціальної бази даних, де опис дій у кожному стані, а також умови переходу в інші стани, задаються у вигляді даних типу *MEMO*, тобто у вигляді рядка символів невизначеної довжини. Модель, по суті, представляє детальну блок-схему алгоритму майбутньої програми у вигляді автоматної моделі. У кожному стані з використанням темпоральної логіки повністю описується послідовність дій, які мають програмно реалізуватися і виконуватися у ньому. Для перетворення такої моделі програми треба виконати повний обхід дерева моделі програми та виконати реалізацію програми на одній із цільових процедурних мов програмування відповідно до вимог технічного завдання. Особливо важливою у цьому підході є можливість створювати паралельні програми шляхом застосування опису спеціальних станів моделі програми, а саме стану-монітору та стану-протоколу. Перший забезпечує доступ до спільних ресурсів кількох процесів, а другий забезпечує паралельне виконання кількох незалежних програмних потоків**.** Таким чином, розроблено новий підхід у створенні надійних, безпомилкових програм шляхом побудови її моделі. При побудові програмної моделі за запропонованим методом вдається уникнути етапу додаткової верифікації цієї моделі, оскільки при коректному описі моделі програми коректною буде і сама програма. При використанні об'єктно-орієнтованого програмування метод дозволяє створювати класи і відповідні програми. Подальшій розвиток запропонованої технології полягає в автоматизації перетворення моделй програм у програми на цільовій мові програмування. Експериментальні застосування цієї технології у Черкаському державному технологічному університеті підтвердили ефективність запропонованої технології для створення надійного програмного забезпечення без помилок та дозволяють рекомендувати її у практичних розробках і впроваджувати у навчальному процесі.

Ключові слова: модель; предикат; темпоральна логіка; невизначений скінченний автомат; процедурна мова програмування.

Голуб Сергій Васильович – д-р техн. наук, проф., зав. каф. програмного забезпечення автоматизованих систем, Черкаський державний технологічний університет, Черкаси, Україна.

Салапатов Володимир Іванович – канд. техн. наук, доц., доц. каф. програмного забезпечення автоматизованих систем, Черкаський державний технологічний університет, Черкаси, Україна.

Немченко Вадим В'ячеславович – канд. техн. наук, доц. каф. програмного забезпечення автоматизованих систем, Черкаський державний технологічний університет, Черкаси, Україна.

Serhii Holub – Doctor of Technical Sciences, Professor, Head of the Automated Systems Software Department, Cherkasy State Technological University, Cherkasy, Ukraine,
e-mail: s.holub@chdtu.edu.ua, ORCID: 0000-0002-5523-6120, Scopus Author ID: 57204158669.

Volodymyr Salapatov – PhD, Associate Professor, Associate Professor at the Software Support of Automated Systems Department, Cherkasy State Technological University, Cherkasy, Ukraine,
e-mail: v.salapatov@chdtu.edu.ua, ORCID: 0000-0001-7567-637X, Scopus Author ID: 6506988429.

Vadym Nemchenko – PhD, Associate Professor at the Software Support of Automated Systems Department, Cherkasy State Technological University, Cherkasy, Ukraine,
e-mail: v.nemchenko@chdtu.edu.ua, ORCID: 0000-0003-2262-719X.