Моделирование и анализ информационных систем. Т. 22, №4 (2015), с. 578–589
Modeling and Analysis of Information Systems. Vol. 22, No 4 (2015), pp. 578–589

DOI: 10.18255/1818-1015-2015-4-578-589

UDC 004.052.42

Automation of Formal Verification of Programs in the Pifagor Language

Ushakova M. S., Legalov A. I.

Received May 18, 2015

Nowadays, due to software sophistication, programs correctness is more often proved by means of formal verification. The method of deduction based on Hoare logic could be used for any programming language and it has the capability of partial automation of the proof process. However, the method of deduction is not widely used for verification of parallel programs because of high complexity of the process. The usage of the functional data-flow paradigm of parallel programming allows to decrease the complexity of the proof process. In this article a proof process of correctness of functional data-flow parallel programs in the Pifagor language is considered. The proof process of a program correctness is considered as a tree where each node is a program data-flow graph, whose edges are marked with formulas in a specification language. The tree root is the initial program data-flow graph with a precondition and a postcondition, which describe restrictions on input variables and correctness conditions of the result of the program execution, respectively. Basic transformations of the data-flow graph are edge marking, equivalent transformation, splitting, folding of the program. By means of these transformations the data-flow graph is transformed and finally is reduced to a set of formulas in the specification language. If all these formulas are identically true, the program is correct. Several modules is distinguished in the system: “Program correctness prover”, “Axioms and theorems library management system” and “Errors analysis and output of information about errors”. According to this architecture, the toolkit for supporting formal verification was developed. The main functionality of the system implementation is considered.

Keywords: functional data-flow parallel programming, Pifagor programming language, programs formal verification, toolkit for supporting formal verification

For citation: Ushakova M. S., Legalov A. I., "Automation of Formal Verification of Programs in the Pifagor Language", *Modeling and Analysis of Information Systems*, **22**:4 (2015), 578–589.

On the authors:
Ushakova Maria Sergeevna, orcid.org/0000-0003-4234-2714, graduate student,
Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo street, Krasnoyarsk, 660074, Russia, e-mail: ksv@akadem.ru

Legalov Alexander Ivanovich, orcid.org/0000-0002-5487-0699, PhD,
Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo street, Krasnoyarsk, 660074, Russia, e-mail: legalov@mail.ru

Introduction

Nowadays there is a tendency to software sophistication that leads to debugging complexity and increase of the system failure risks. As the result, methods of program verification, particularly formal verification, started to develop rapidly. *Formal verification* is a proof of programs correctness by finding a correspondence between the program and its specification, which describes the aim of the development [1]. The correspondence between the program and its specification is established by the rigorous mathematical proof. The main advantage of formal verification is the ability to prove the absence of errors in the program formally.

The method of deduction based on Hoare logic [2, 3] is the most universal method of formal verification. Hoare logic is an extension of a formal system $\mathfrak{I}$ with certain formulas called Hoare triples. A *Hoare triple* is a program, namely the source code, and two formulas of the theory $\mathfrak{I}$, which describe restrictions on input variables and correctness conditions of the result of the program execution. These formulas are called *precondition* and *postcondition*, respectively. The extended formal system is distinguished from $\mathfrak{I}$ by additional axioms and inference rules, which allow to deduce certain program properties, particularly the program correctness. The program is correct if its Hoare triple is identically true. So the main idea of this approach is to derive a formula of formal system $\mathfrak{I}$ from the Hoare triple and then prove the truth of this formula within the formal system $\mathfrak{I}$.

The described method could be used for any programming language. Its main advantage is the capability of partial automation of the proof process. It should be pointed out that axioms and inference rules are individual for each programming language, since they are generated on the base of the language semantics. So each programming language has its individual corresponding formal system, therefore the difficulty of the proof process is also different.

Nowadays there are some achievements in practical use of the deduction method for verification of sequential imperative programs. There exist several systems that aid the proof process, for example, Boogie [4] and Spectrum [5] for programs in the C language, and LOOP [6] and KeY [7] for programs written in an object-oriented language. Boogie generates verification conditions for the theorem prover Simplify [8], and LOOP uses the interactive theorem prover PVS [9], for which it generates proof obligations that look like Hoare triples.

The development of formal verification methods are topical especially for parallel programming. Formal verification complexity for parallel imperative programs increases rapidly in comparison with sequential ones. The main problem is system resource conflicts, for example the misuse of shared memory or deadlocks of processes in distributed memory.

An alternative to imperative programming is the functional data-flow paradigm and its implementation the Pifagor (Parallel Informational and Functional AlGORthmic) language [10], [11]. This language allows to exclude resource conflicts. Each program is a function, so in Pifagor there are neither variables nor loops, and function execution starts only on data readiness. Another distinguishing feature of this language is the ability to achieve the maximum parallelism of the program, as parallelism is implemented at the level of operations. So after formal verification the correct program could be transferred to a system with specific architecture and limited resources, with the reduction of

parallelism if needed.

Hoare logic for the Pifagor language, which allows to prove programs correctness, has been already considered [12], [13]. First-order logic is used as the specification language to specify preconditions and postconditions. However the proof process is rather tedious, as proving the truth of a single triple is usually reduced to proving the truth of several transformed triples. The necessity to take into account a great number of triples makes the proof process quite complicated. So the aim of this work is to develop a toolkit for supporting the formal verification of functional data-flow parallel programs.

1. Transformations of a Labeled Data-Flow Graph

A program written in Pifagor is more demonstrable when represented as a *data-flow graph*. It is an acyclic directed graph that represents the data-flow of the program. The nodes of this graph represent program operators and the edges represent channels of data-flow between incident nodes [11]. Let us call a data-flow graph, whose edges are marked with formulas in the specification language, as a *labeled data-flow graph* (LDFG). If in a data-flow graph only the input and output edges are marked, then this graph corresponds to the Hoare triple. In this triple the program is represented by the graph, the precondition is the formula that marks the input edge and the postcondition is the formula that marks the output edge. Let us define the following transformations of a labeled data-flow graph:

1) edge marking;
2) modification of a data-flow graph:
   (a) equivalent transformation,
   (b) splitting;
3) folding of the program.

The process of proving the correctness of functional data-flow programs could be considered as a sequence of transformations of the initial labeled data-flow graph. The "initial labeled data-flow graph" is the LDFG that corresponds to the initial Hoare triple, namely the graph in which the input and output edges are marked with the user-defined precondition and postcondition respectively. Sequential transformations of the initial LDFG result in the set of *fully marked LDFG*. Each edge of a fully marked LDFG is marked with a singe formula. By the folding transformation these graphs are transformed into Hoare triples, which could be directly transformed into formulas in the specification language. The latter formulas are checked for truth. If all derived formulas are identically true, then the initial Hoare triple is also identically true, and the program is correct.

1.1. Edge Marking

The principle of marking graph edges with formulas is described in [12], [13]. A formula, attached to the edge of the graph, describes properties of data transferred through this edge.

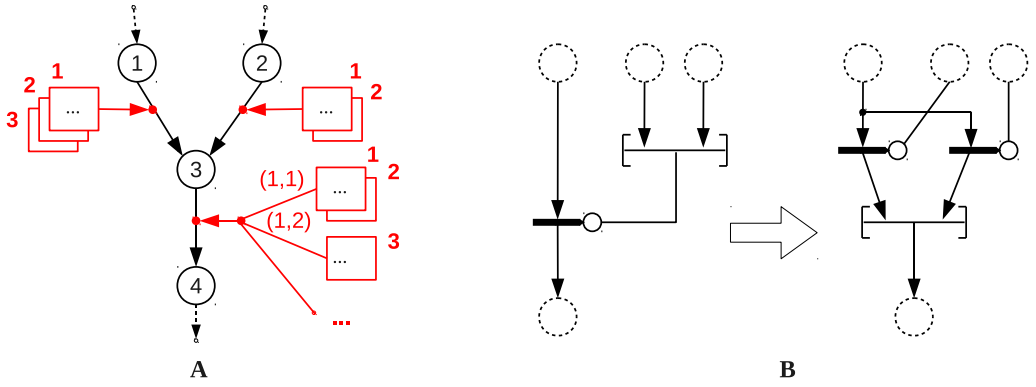

Fig. 1. The scheme of the labeled data-flow graph transformations. A — a part of a program data-flow graph marked with formulas. The graph nodes (denoted by circles) are program operators, the edges are data-flow links and are marked with formulas (denoted by rectangles); numbers near the formulas are formula indices, and parent formula indices are in parentheses. B — the equivalent transformation called the "parallel list release". The parallel list is denoted by a horizontal line in square brackets, the operator of interpretation is denoted by a bold horizontal line with a circle (the circle indicates a function input).

Edges of a data-flow graph are marked on basis of axioms (for built-in functions of the Pifagor language) and theorems (for user-defined functions with proved correctness). Several axioms or theorems could describe one function, so after edge marking several new graphs are derived.

Let us introduce the relation of hierarchy on the set of formulas. A formula attached to an edge (a, b) is called parent to a formula attached to an edge (b, c) for graph nodes a, b, c.

Edges marking does not change the data-flow graph of the program, so if it gives several new LDFGs, then they differ between each other only in one formula of the marked edge. That is why for the sake of compactness these LDFGs could be united into a single LDFG with the edge marked with several formulas simultaneously. Then the relation of hierarchy between formulas allows us to split the compact representation into the initial LDFGs.

Let us illustrate the compact representation of LDFG by means of the following example. Consider a graph of some program, depicted in the figure 1.A. The four operators of this program are connected with data-flow links. The node 3 receives input data from the nodes 1 and 2, the node 4 receives data from the node 3. Initially, the edges of the graph are not marked.

At first let us mark the edge $(1, 3)$. If some three axioms for the operator 1 are suitable for transforming the graph, then the result of the transformation gives us three new LDFGs, whose compact representation is equal to the initial graph with edge $(1, 3)$ marked with three formulas simultaneously. Then let us mark the edge $(2, 3)$. The number of suitable axioms for each of three graphs, obtained on the previous step, is the same, since the operators 1 and 2 are independent. If there exist two axioms for the operator 2

suitable for the transformation of these three DFMGs, then the result of transformation gives $6 = 3 \cdot 2$ new graphs. Their compact representation is obtained from the compact representation from the first step by marking the edge $(2, 3)$ with two new formulas. The operator 3 depend on the operators 1 and 2, so different number of axioms could appear applicable to each of the six graphs. Let this number be equal to two for the first LDFG (so its transformations gives two new LDFGs) and one for the rest five LDFGs. Thus, we obtain $7 = 1 \cdot 2 + 5 \cdot 1$ graphs after the transformation. This number equals the total number of suitable axioms as well as the number of formulas attached to the edge $(3, 4)$ of the compact representation. In the indices (i, j) of the formulas, marking the edge $(3, 4)$ in the figure 1, i is the index of the formula attached to the edge $(1, 3)$, parent to $(3, 4)$, and j is the index of the formula attached to the edge $(2, 3)$, also parent to $(3, 4)$.

1.2. Modification of a Data-Flow Graph

The second type of LDFG transformations changes the program graph. This equivalent transformation is done according to the rules of equivalent transformations for operators and links of the Pifagor language (see [10], [14] for details). The result of such transformations is a LDFG with a modified graph. The example of the equivalent transformation called the "parallel list release" is given in the figure 1.B. A parallel list allows to declare explicitly that all its input operators could be executed simultaneously. In the left graph the parallel list of two elements is the functional input of the interpretation operator. An operator of interpretation has two inputs: one for an argument and another for a function. The operator applies the function to the argument. In the equivalent graph on the right each element of the initial parallel list is the functional input of its own operator of interpretation, and the result of both interpretations is transfered to the parallel list.

Another transformation of the second type is splitting. Application of this transformation to a single LDFG results in two or more LDFGs with modified graphs. Splitting could be used for proof simplification, for example, if we know all the choices in the "select list element" operator. Thus it is possible to simplify not only the data-flow graph, but also marking formulas.

1.3. Folding

The third type of transformations is folding of the program. The folding transformation is applied to the marked edge (except the input and output edges of the program). The whole induced subgraph with nodes, from which the beginning of the concerned edge is reachable (except the input argument), is replaced by a new variable, and formula attached to the concerned edge is added to the precondition of the initial LDFG. The program becomes "shorter" when a part of the code is replaced by a variable. The folding transformation could be applied when graph is completely or partly marked with formulas. If folding is applied to the all marked edges of the graph, then there is a Hoare triple equal to the resulting LDFG. Let us call such folding as *complete folding*. When a LDFG has all edges marked, complete folding transforms its graph into a single variable. The properties of this variable are described in precondition, which also describes the properties of all the data transfered through the edges of the initial graph. At the same time this variable is the return value of the program. Such program of one variable

is called the *empty program*. To prove its correctness it is sufficient to show that the precondition implies the postcondition.

As the result we could make the following conclusion: the whole proof process could be represented as a tree whose root is the initial LDFG, child nodes are derived from parent nodes by one of the transformations 1)–3), and leaves are fully marked LDFGs which are transformed into formulas by the complete folding. Let us call such tree as a *proof tree* or a proof of program correctness.

2. Architecture of the Toolkit

A general scheme of the system for supporting formal verification of functional data-flow parallel programs is given in the figure 2.

Several modules could be distinguished in the system: "Program correctness prover", "Axioms and theorems library management system" and "Errors analysis and output of information about errors". The "Formula prover" module (formula verifier) is isolated from the system as it is a third-party application and could be excluded from the proof process, in this case its operations should be done by a user (manually or with the help of a more convenient verifier). More information about provers could be found in [15, 16, 17, 18].

The principle of the system operation is the following. A user passes a program in the Pifagor language and program specification in the specification language to the system. The "Program correctness prover" module generates the initial DGFM (which is equal to the initial Hoare triple) and starts the proof process, i.e. marking edges with formulas. Axioms and proved theorems from the "Axioms and theorems library" are used in this process. The "Program correctness prover" module sends requests to the "Axioms and theorems library management system". In case the requested function is not present in the library, the error of marking impossibility is returned, which is processed by the "Errors analysis" module. If axioms (theorems) for the considered function are present in the library, then the "Program correctness prover" module makes a selection among them. For each axiom (theorem) available for the considered function the applicability condition in the specification language is formed. This condition is passed to the formula verifier, and if it is true or satisfiable then the axiom (theorem) is used for marking. Otherwise the axiom (theorem) is rejected. After all edges in the data-flow graph are marked, the complete folding is performed. Hoare triples corresponding to the obtained LDFGs are transformed into formulas (let us call them *final formulas*) and are passed to the formula verifier to check their truth. If all the formulas are identically true, then the program is correct and its initial Hoare triple is a theorem. The "Program correctness prover" module passes this theorem (together with the proof) to the "Axioms and theorems library management system" to save the theorem in the library. If the truth of the formula is not proved then the "Errors analysis" module detects the error and informs the user.

Let us consider the operation of the "Program correctness prover" module in detail (see the figure 2). The "Proof process control" block is the main one, it interacts with the user, takes his commands and visualizes the proof process. This block forms the proof tree. Having received a LDFG this block searches operators with unmarked edges

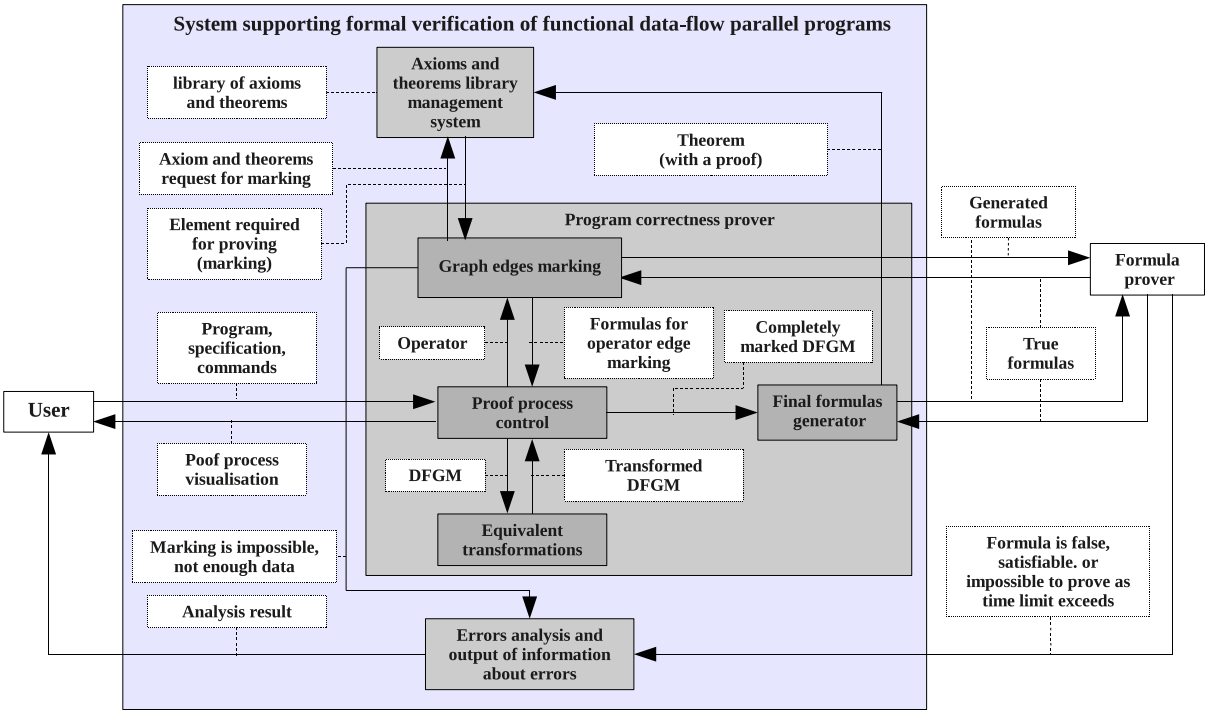

Fig. 2. General scheme of the toolkit for supporting formal verification of functional data-flow parallel programs.

and passes them, firstly, to the "Equivalent transformations" block and, secondly, to the "Graph edges marking" block, which generates marking formulas. After the LDFG is fully marked it is passed to the "Final formulas generator" block that performs the complete folding and generates the set of final formulas.

The considered system could work in several modes:

1) completely manual;

2) partially automated;

3) automated.

In the first case the user uses only the "Proof process control" and "Final formulas generator" blocks and also the "Errors analysis" module. The user is to mark the edges with formulas and prove the truth of the final formula manually. In the second case only the "Formula prover" module is not used. And in the automated mode all modules are used.

3. System Implementation

According to the architecture considered above, the toolkit for supporting formal verification of functional data-flow parallel programs was developed. This toolkit allows to construct a proof tree of programs in the Pifagor language.

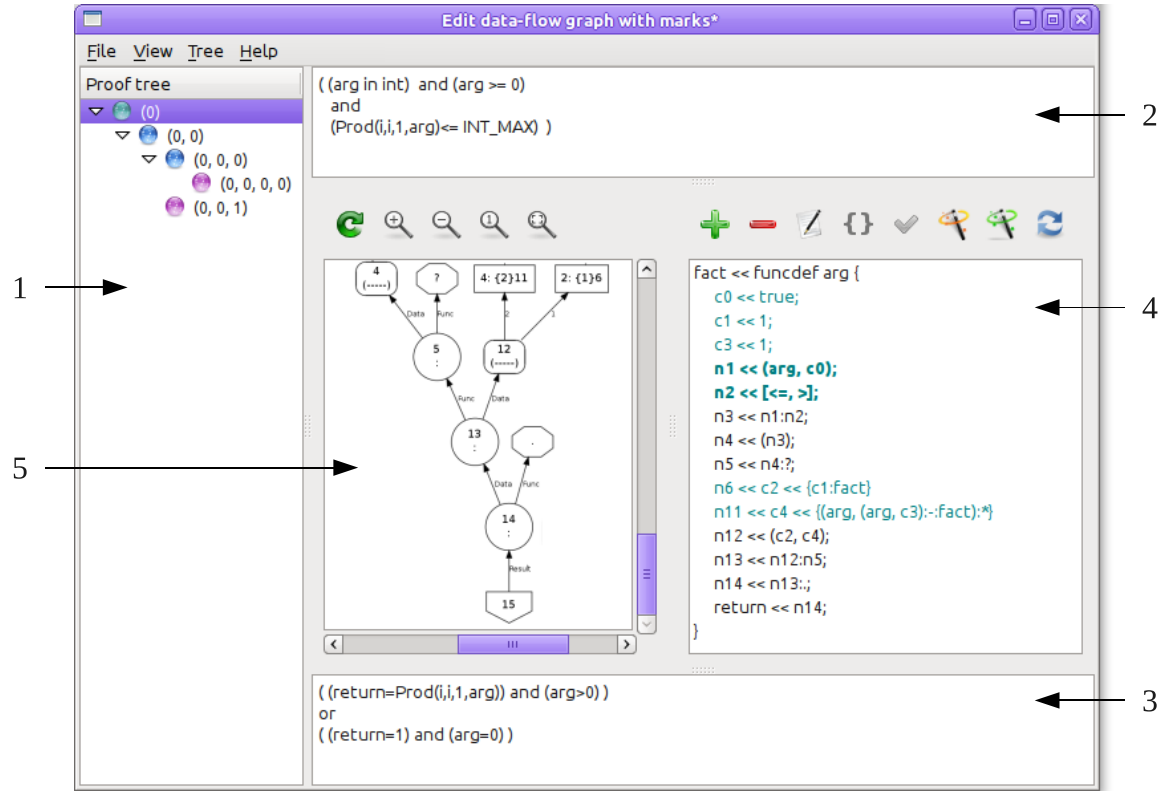

Fig. 3. The main window of the toolkit for supporting formal verification of functional data-flow parallel programs; 1 — the proof tree nodes editor, 2 — the precondition entry field, 3 — the postcondition entry field, 4 — the program code editor, 5 — the graphical representation of the reversed data-flow graph of the program

The system has a graphical user interface for editing a proof tree. As an input data, the system receives a program in the reversed data-flow graph (RDFG) format [19, 20], which represent data dependencies in the program (this graph is the same as the data-flow graph of the program except the edges that has the opposite direction), and the formulas marking its edges. Also a previously saved LDFG or a proof tree could be loaded.

The main window of the system (shown in the figure 3) is the proof tree editor. On the left part it has a tree nodes editor and on the rigth it has the LDFG editor, which shows the current tree node. The proof tree editor allows to insert new or already existing LDFGs, copy and delete current LDFGs and insert a copied LDFG as a child to the current LDFG.

The LDFG editor has precondition and postcondition entry fields in its top and bottom parts respectively. The graphical representation of the reversed data-flow graph of the program is on the left part of the editor and the program code editor is on the right. A third-party program GraphViz is used to generate the graphical representation of the RDFG. This program uses DOT files (DOT is a plain text graph description language). The program code editor has a tool bar and a text box for a program source code. The source code of a program is divided into lines, where each line is equal to a

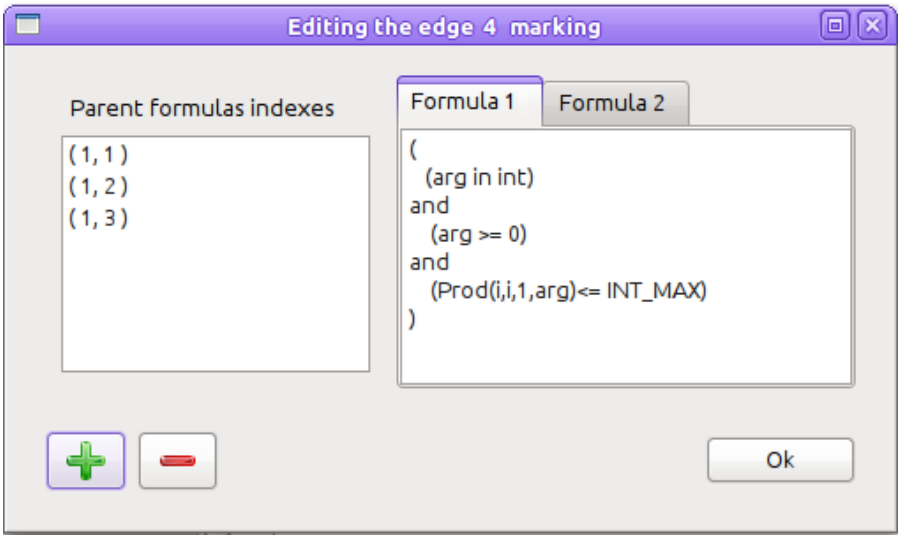

Fig. 4. The window to edit formulas hierarchy for the marking edge

single node of the program RDFG. The RDFGs are in a text format and are created according to the program code. Such RDFG text format has the right order of function calls, which means that a function is called only after its definition. So this RDFG text format allows to restore the program code unambiguously.

The program code editor allows to insert, delete and edit RDFG nodes. At the same time certain restrictions do not allow the user to use an operand before its definition. The editor is also capable of editing delay lists. The operator of grouping into the delay list (briefly, the delay list) contains a subgraph of the program. Operators from the delay list are not executed even if all their arguments are ready. Their execution starts only after the delay release, when the delayed subgraph becomes a part of the whole graph [10], [11]. The program code editor allows to add any operator to the delay list together with all operators that are used by the concerned operator. A delay list is treated as a constant, so editing delayed operators is possible only after the delay release.

Edges marking could be done manually, by means of the LDFG editor, or by calling the function of automatic edges marking. The LDFG editor uses the compact representation of LDFGs, which was described above. If an operator output edge was not marked then the readiness of marks for all operator input edges is checked. If some graph in the compact representation has an unmarked edge, then error message of marking impossibility for the current edge is returned to the user. If all input edges are marked then the window (see figure 4) with formulas hierarchy for the output edge of the current node is shown. All formulas of the edge are divided into groups according to different combinations of their parent nodes formulas. There is a list of parent formulas indices on the left part of the window. Marking formulas of the edge currently being marked are displayed on the pages of the tab widget on the right. The user could add new formulas, and the system automatically changes the indices of all child nodes (if they are already marked) and the necessary number of empty pages on the tab widget are added for new formulas. If a formula is deleted, all child formulas for the current formula are also deleted, and indices of the rest formulas are changed respectively.

Also available are the following functions: autotransforming, LDFG automarking and

final formulas generating.

The autotransforming of the current LDFG (namely, equivalent transformations that are performed automatically) is done in case the current LDFG is a leaf of the proof tree, otherwise its child subtree should be deleted. Furthermore, only *ready* operators, whose all input edges are marked and an output edge is not marked, are transformed. The applicability of the equivalent transformation is checked for all ready operators. If the equivalent transformation is applicable to a LDFG, then a new obtained LDFG becomes a child of the initial LDFG in the proof tree.

The function of LDFG automarking is applied to the current graph. At first equivalent transformations are applied, then a ready operator is searched to perform the marking of its output edge. If the found operator is a list, then the output edge is marked with the constant *true*. If it is the operator of interpretation, then a request for a list of axioms (for built-in functions) or a list of theorems (for user-defined functions) is send to the library of axioms and theorems. If the function is present in the library, then for each axiom (theorem) the applicability condition is formed. The truth or satisfiability of this condition should be checked manually. If this condition is identically false for some axiom or theorem, then it is rejected. The remaining axioms and theorems are used for marking the edge of the considered operator of interpretation with several formulas. Further, another ready operator is searched.

If all edges of current LDFG are marked, then the complete folding is applied. The result is a set of Hoare triples with the empty program. They are transformed into formulas by the final formulas generating function. The user should check whether all these formulas are identically true to prove the program correctness.

Thus, the completely manual and partially automated modes of the system described in the previous section have already been implemented.

4. Conclusion

This paper describes main transformations of a labeled data-flow graph during the program correctness proving. Main concepts of the architecture of the toolkit for supporting formal verification of functional data-flow parallel programs were developed. The task of implementing the toolkit is partially solved. At present the work is underway to enhance the partially automated mode functionality and implement the automated mode.

References

[1] Nepomnyaschiy V. A., Ryakin O. M., *Prikladnyie metodyi verifikatsii programm*, Radio i svyaz, Moscow, 1988, 255 pp., [in Russian].

[2] Hoare C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, **10**:12 (1969), 576–585.

[3] Floyd R. W., "Assigning meaning to programs", Mathematical Aspects of Computer Science, ed. J. T. Schwartz, 1967, 19–32.

[4] Barnett M., Chang B. Y. E., Deline R., et al., "Boogie: A Modular Reusable Verifier for Object-Oriented Programs", LNCS, **4111**, 2006, 364–387.

[5] Nepomniaschy V. A., Anureev I. S., Atuchin M. M., "C Program Verification in SPECTRUM Multilanguage System", *Automatic Control and Computer Sciences*, **45**:7 (2011), 413–420.

[6] Van den Berg J., Jacobs B., "The LOOP compiler for Java and JML", LNCS, **2031**, 2001, 299–312.

[7] Ahrendt W., Baar T., Beckert B., "The KeY Tool", *Software and System Modeling*, **4**:1 (2005), 32–54.

[8] Detlefs D., Nelson G., Saxe J. B., "Simplify: a theorem prover for program checking", *Journal of the ACM*, **52**:3 (2005), 365–473.

[9] Owre S., Rajan S., Rushby J. M., "PVS: Combining Specification, Proof Checking, and Model Checking", LNCS, **1102**, 1996, 411–414.

[10] Legalov A. I., "Funktsionalnyiy yazyik dlya sozdaniya arhitekturno-nezavisimyih parallelnyih programm", *Vyichislitelnyie tehnologii*, **10**:1 (2005), 71–89, [in Russian].

[11] Legalov A. I., Kuzmin D. A., Kazakov F. A., "Na puti k perenosimyim parallelnyim programmam", *Otkryityie sistemyi*, **5** (2003), 36–42, [in Russian].

[12] Kropacheva M. S., Legalov A. I., "Formal Verification of Programs in the Functional Data-Flow Parallel Language", *Automatic Control and Computer Sciences*, **47**:7 (2013), 373–384.

[13] Kropacheva M. S., Legalov A. I., "Formal Verification of Programs in the Pifagor Language", *Parallel Computing Technologies (PaCT-2013)*, 12th International Conference (September 30 - October 4, 2013. Saint-Petersburg, Russia), LNCS, **7979**, 2013, 80–89.

[14] Kropacheva M. S., "Formalizatsiya semantiki funktsionalno-potokovogo yazyika parallelnogo programmirovaniya Pifagor", *Problemyi informatizatsii regiona (PIR-2011): Materialyi XII Vserossiyskoy nauchno-prakticheskoy konferentsii* (Krasnoyarsk, 22 – 23 noyabrya 2011), Publishing of the Siberian Federal University, Krasnoyarsk, 2011, 144–148, [in Russian].

[15] *The Seventeen Provers of the World*, AI Systems, LNAI, **3600**, ed. Wiedijk F., 2006, 169 pp.

[16] Gordon M., Melham T., *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[17] Bertot Y., Casteran P., *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004, 494 pp.

[18] Nipkow T., Paulson L., Wenzel M., *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS, **2283**, 2002, 205 pp.

[19] Legalov A. I., Savchenko G. V., Vasilev V. S., "Sobyitiynaya model vyichisleniy, podderzhivayuschaya vyipolnenie funktsionalno-potokovyih parallelnyih programm", *Sistemyi. Metodyi. Tehnologii*, **1**:13 (2012), 113–119, [in Russian].

[20] Matkovskiy I. V., "Parallelnaya sobyitiynaya mashina dlya funktsionalno-potokovogo yazyika "Pifagor"", *Informatsionnyie i matematicheskie tehnologii v nauke i upravlenii: Sbornik trudov XVVII Baykalskoy Vserossiyskoy konferentsii s mezhdunarodnyim uchastiem* (Irkutsk – Baykal, 30 iyunya – 9 iyulya 2012), **2**, Publishing of the Melentiev Energy Systems Institute of Siberian Branch of the Russian Academy of Sciences, Irkutsk, 2012, 186–193, [in Russian].

DOI: 10.18255/1818-1015-2015-4-578-589

Автоматизация формальной верификации программ на языке Пифагор

Ушакова М.С., Легалов А.И.

получена 18 мая 2015

В связи с увеличением сложности программного обеспечения корректность программы всё чаще доказывается с помощью методов формальной верификации. Дедуктивный анализ на основе исчисления Хоара применим для произвольных языков программирования и допускает частичную автоматизацию процесса. Однако дедуктивный анализ не нашёл широкого применения для верификации параллельных программ из-за высокой сложности процесса. Использование функционально-потоковой парадигмы параллельного программирования позволяет снизить сложность доказательства. В работе рассматривается процесс доказательства корректности функционально-потоковых параллельных программ на языке Пифагор и предлагается архитектура инструментального средства для поддержки процесса доказательства. Процесс доказательства корректности программы представляется в виде дерева, каждый узел которого — информационный граф программы, в котором дуги размечены формулами на языке спецификации. Корнем дерева является информационный граф программы с предусловием и постусловием, которые описывают ограничения на входные переменные и условия корректности результата работы программы соответственно. Основные преобразования, применимые к информационному графу программы: разметка дуг, эквивалентное преобразование, расщепление, свертка программы. Посредством данных преобразований информационный граф модифицируется и, в конечном счете, сводится к набору формул на языке спецификации, истинность которых будет свидетельствовать о корректности программы. Предложена архитектура системы поддержки процесса доказательства, которая позволяет строить дерево доказательства. В системе выделено несколько основных модулей: «Модуль доказательства корректности программы», «Система управления библиотекой аксиом и теорем» и «Модуль анализа ошибок и выдачи информации об ошибках». Согласно описанной архитектуре, разработано инструментальное средство для поддержки формальной верификации, которая позволяет строить дерево доказательства. Описана основная функциональность реализации системы.

Ключевые слова: функционально-потоковое параллельное программирование, язык программирования Пифагор, формальная верификация программ, инструментальные средства для поддержки формальной верификации

Для цитирования: Ушакова М.С., Легалов А.И., "Автоматизация формальной верификации программ на языке Пифагор", *Моделирование и анализ информационных систем*, **22**:4 (2015), 578–589.

Об авторах:
Ушакова Мария Сергеевна, orcid.org/0000-0003-4234-2714, аспирант,
Институт космических и информационных технологий, Сибирский федеральный университет,
ул. Академика Киренского, 26, г. Красноярск, 660074 Россия, e-mail: ksv@akadem.ru

Легалов Александр Иванович, orcid.org/0000-0002-5487-0699, д-р техн. наук, профессор, заведующий кафедрой вычислительной техники,
Институт космических и информационных технологий, Сибирский федеральный университет,
ул. Академика Киренского, 26, г. Красноярск, 660074 Россия, e-mail: legalov@mail.ru