COMPUTING

SYSTEM OF PROGRAMS PROVING

Alexander Letichevsky, Olexander Letichevskiy,
Marina Morokhovets, Vladimir Peschanenko

Abstract*: The paper is devoted to the methods of programs proving in the Insertion Modeling System IMS. Architecture and functional possibilities of IMS, the main notions of insertion modeling were presented in this work. Floyd's insertion machine and the methods of the satisfiability checking, and the usage of those methods for programs proving were described in the paper.*

Keywords*: multi-agent systems, insertion modeling, program proving.*

ACM Classification Keywords: *D.2 SOFTWARE ENGINEERING D.2.4 Software/Program Verification.*

Introduction

The system of programs proving - a new and modern system programs proving that is designed to maintain a high level of training of qualified specialists in the field of programming. This system is designed on the basis of insertion modeling system IMS [IMS, 2011], the algebraic programming system APS [APS, 1989], developed in the last century at the Glushkov Institute of Cybernetics [CYB, 2012], with the participation of authors of Kherson State University [KSU, 2012] under the leadership of prof. Alexander Letichevsky.

Ukraine has a high potential for training in information technologies. These specialists are appreciated throughout the world, primarily because a system of basic training in information technologies, especially of software developers, is created in our country. However, their training lags behind the world level, because it is very important not only to write programs, but to verify, improve reliability and efficiency. To do this, for example, Microsoft has created a new system of Spec # [SPEC, 2012], which is connected to the development of the highest qualification specialists: mathematicians, engineers, etc.

Recently the idea of using high-tech industries of software, which proved correct, has become increasingly popular in the world. However, in the software market of Ukraine and abroad the software products able to prove the correctness of programs are absent. This is due to the fact in order to create such software it is required very qualified specialists in sphere of technologies, as well as in basic sciences. In addition, when mastering training courses of relevant disciplines, students don't have opportunities to learn proving programming with use of modern software. The use of such system of proving programming in higher educational institutions of Ukraine may significantly improve the quality of basic training of our programmers.

The article is devoted to proving methods of program correctness in insertion modeling system IMS. The section "Possibilities – Present and Future" describes the present possibilities of our system and shows the future opportunities of it. The section "Insertion modeling" provides basic information about the insertion modeling, and describes the architecture and functionality of the Insertion Modeling System IMS. The following section

describes the architecture of the insertion machine system IMS, developed for the validation of programs and based on the method of Floyd annotation programs. In the section "Tools for checking the satisfiability of formulae" attention is paid to the methods of checking the satisfiability of formulae that arise in the process of correctness proving of programs using Floyd. In the section "Example of programs proving" describes some applications of the method of Floyd insertion machine to verify imperative program.

Possibilities – Present and Future

Program's proving system should ideally consist of the following modules: Preparation module, Verification module, Dialog module, Knowledge of verification module.

The functionality of Preparation module:

- creates an environment for the annotated program;
- proposes set of various kinds of annotated examples of the programs;
- configures the subject area by type of verifiable programs (sequential, parallel), in the form of assertions in the annotations to be checked (approval of the integers, real numbers etc), by type of structured objects (defined by a set of tools required for verification);
- configures the subsystem verification.

The functionality of Verification module:

- proves the statements of annotations (using internal and external tools of the system);
- proves and checks satisfiability of statements in various subject areas.

The functionality of Dialog module:

- provides opportunities to follow the process of verification, giving hints;
- prepares a report on verification with varying degrees of details;
- allocates errors in the program;
- gives recommendations for program improving (if the result of the verification program is not recognized as correct).

The functionality of Knowledge of verification module:

- stores annotated examples of programs of various kinds;
- stores general recommendations for creation of annotations;
- stores other background information (syntax of annotation, etc).

This article discusses the first proving programming system version including only the Verification module. Other modules will be discussed in the future publications of the authors.

Insertion modeling

Insertion modeling - an approach to modeling complex distributed systems based on the theory of interaction of agents and environments [Letichevsky, 1996]. Mathematical foundations of this theory have been presented in [Letichevsky, 1998]. During the last decade insertion simulation was used to verify the software requirements [Letichevsky, 2008]. Theory of Interaction of agents and environments has been proposed as an alternative to known theories of interaction, such as Milner's CCS [Milner, 1989] and the π-calculus [Milner, 1999], CSP Hoare [Hoare, 1985] and mobile ambient Cardelli [Cardelli, 1998], etc. The idea of decomposition of the system and

presenting it as a composition of the medium and agents that are immersed in this environment is implicit in all theories of interaction.

Another source of ideas for the insertion modeling is to find universal programming paradigms (i.e. ASM Gurevich [Gurevich, 1995], the universal theory of programming Hoare [Hoare, 1998], rewriting logic Meseguer [Meseguer, 1992]). These ideas have been adopted as the basis for insertion modeling system IMS [IMS, 2003], developed as an extension of the algebraic programming system APS [APS,1989]. The first version of the IMS system and some simple examples of its use can be found in [APS&IMS, 2012]. IMS has many successful applications, one of these applications, the focus of this work, the proving of programs correctness in the insertion modeling.

Floyd's insertion machine. The study process validation program has a long history that began with the work of Hoare [Hoare, 1969] and Floyd [Floyd, 1967]. Conditions for the correctness of the program are of the form $\alpha \rightarrow < P > \beta$ or $\alpha \rightarrow [P]\beta$. Formula α and β (pre-and post-condition) are the formulae of the language specification (i.e., language, first order predicate calculus), ***P*** - the specification of the program (it is assumed that the initial state of memory is given). The first formula means if the condition α is valid and the program ***P*** terminates, then the condition β is also valid in the final state of memory. The second formula β is correct and complete and it means that if the condition α is valid, then the program ends and the condition β is also valid.

In the current implementation of the system, we consider the reduced version of the analytical insertion machine (it is designed for the analysis of the model, check its properties, etc.) based on the annotation of programs by Floyd.

Usually, insertion function is denoted as $E[u]$ were ***E*** is the state of environment and u is the state of an agent (agent in a given state). $E[u]$ is a new environment state after insertion an agent u. So, the expression $E[u[v], F[x, y, z]]$ denotes the state of a two level environment with two agents inserted into it. At the same time E is an external environment of a system $F[x, y, z]$ and ***F*** is an internal environment of it. All agents and environments are labeled or attributed transition systems (labeled systems with states labeled by attribute labels [Letichevsky, 2008]). The states of transition systems are considered up to bisimilarity, denoted as $\sim_B$. This means that we should adhere to the following restriction in the definition of states: if $E \sim_B E'$ and $u \sim_B u'$ then $E[u] \sim_B E'[u']$ ($\sim_B$ denotes bisimilarity).

The general architecture of insertion machine is presented on fig. 1.

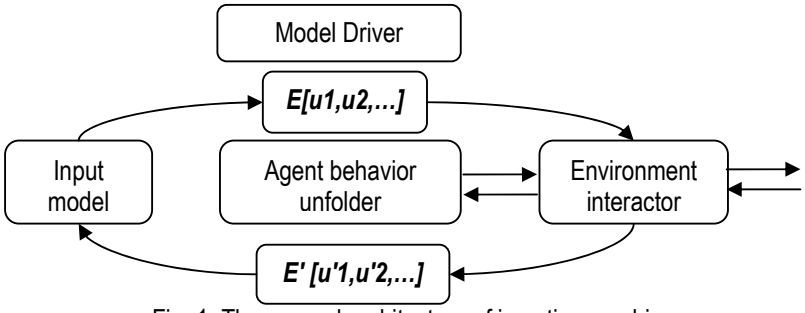

Fig. 1. The general architecture of insertion machine

The main component of insertion machine is model driver, the component which controls the machine movement along the behavior tree of a model. The state of a model is represented as a text in the input language of insertion machine and is considered as an algebraic expression. The input language of *Input model* includes the recursive definitions of agent behaviors, the notation for insertion function, and possibly some compositions for environment states. Before computing insertion function the state of a system must be reduced to the form $E[u_1, u_2, ...]$. This

functionality is performed by the module called *Agent behavior unfolder*. To make the movement, the state of environment must be reduced to the normal form

$$\sum_{i \in I} a_i.E_i + \varepsilon$$

where a_i are actions, E_i are environment states, ε is a termination constant. This functionality is performed by the module *Environment interactor*. It computes the insertion function calling if it is necessary the *Agent behavior unfolder*. If the infinite set I of indices in the normal form is allowed, then the weak normal form $a.F + G$ is used, where G is arbitrary expression of input language.

The insertion function of Floyd's insertion machine is the permitting function for sequential or parallel insertion. Formulae of predicate calculus are environment state. The transition relation of system is presented by the following rules:.

$$\varphi(p) \xrightarrow{ask\,u} (\varphi \wedge [p]u), Sat(\varphi \wedge [p]u)$$
$$\varphi(p) \xrightarrow{x:=y} \varphi[p^*(x := y)]$$
$$\varphi(p) \xrightarrow{go\ to\ L} \varphi[p, model.L]$$
$$\varphi(p) \xrightarrow{assumption(\psi)} (\varphi \wedge [p]\psi)[p]$$
$$\varphi(p) \xrightarrow{assertion(\psi)} \psi[empty], \neg Sat(\varphi \rightarrow [p]\psi)$$
$$\varphi(p) \xrightarrow{assertion(\psi)} 0, Sat(\varphi \rightarrow [p]\psi)$$
$$\varphi(p) \xrightarrow{stop} \varphi[Delta]$$

Here p is a parallel assignment or Δ (successful termination state). Function $Sat(u)$, which checks satisfiability of formula u, is used for permitting conditions. Conditional operator *If u then P else Q* is considered as functional expression and is translated by unfolder by means of the rule:

$$If\ u\ then\ P\ else\ Q = ask\ u.P + (ask \neg u).Q$$

Loops operators and other programming constructions could be introduced in a similar way. The Floyd's machine could prove the partial correctness of nondeterministic program, because nondeterministic choice is an initial operation of an input language of the system. If unfolding for parallel composition is defined, then Floyd's machine could prove a partial correctness of parallel programs over shared memory.

Tools for checking the satisfiability of formulae

The system uses two different kinds of algorithms: internal and external. Three constants are used for resulting of each satisfiable algorithm:

- *proved* means that a input formulae is satisfiable;
- *refuted* means that a input formulae is not satisfiable;
- *not proved* means that an input formula can't be checked by algorithm because of some restrictions (non-linearity of the formula - a formula in which the variables are found with degree more than one, etc).

The general algorithm of checking satisfiability of formulae looks like the next sequences of rules:

- *it is used first internal algorithm.* If it returns *proved*(*refuted)* then returns *proved*(*refuted),* if it returns *not proved* then uses external algorithm.
- *it is used external algorithm.* If it returns *proved*(*refuted)* then returns *proved*(*refuted),* if it returns *not proved* then uses the next external algorithm if it exists, and if not then asks user for answer.

Internal satisifiability checking algorithm. First, superpositions of functional expressions are eliminated by successive substitution of every innermost occurrence of $f(x)$ by a new variable y, bound with an existential quantifier, and adding the formula $y = f(x)$. For example, formula $P(f(g(x)))$ is replaced by formula $\exists y((y = g(x)) \wedge P(f(y)))$. After all of such replacements, there will no more nested functional expressions. For every attribute expression f of array or functional type, all its occurrences $f(x_1), f(x_2), \ldots$ with different parameters $x_1, x_2, \ldots$ are considered: $f(x_i)$ is replaced by variable y_i, bound with an existential quantifier, and equations $(x_i = x_j) \rightarrow (y_i = y_j)$ are added. At this point, there will be only simple attributes and the method for simple attributes is applied. Elimination of functional expressions imposes restrictions on the range of values of arguments for the functional attributes of type array with integer or enumerated indexes. For example, if the attribute f has a type $array(m, \tau)$, where m is a number, and τ is an enumerated type with constants $a_1, a_2, \ldots$, then during elimination of functional expression $f(i, u)$, the generated formula will include conjunctive constraints $0 \le i \wedge i \le m - 1 \wedge (u = a_1 \vee u = a_2 \vee \ldots)$.

The result is a closed formula (i.e. a formula not containing attributes) and all bound variables have types integer, real, or symbolic, or are enumerated types.

The deductive system of IMS contains three specialized provers: an integer prover for Pressburger arithmetic, the Furie-Motskin algorithm for linear arithmetic over reals, and a symbolic prover that includes an algorithm of finding the most general solution for a system of symbolic equations (modified Montanari-Rossi algorithm of unification integrated with numerical provers). More details about internal satisfiable algorithm are in [SAT, 2012]

External satisifiability checking algorithms. We provide the next interface for developers to integration of external tools for satisfiable checking:

- Three specific constants are implemented in class Clew[APS,1989]: *proved*, *refuted*, *not_proved* (this class is a set of functions for working with tree).
- The function *put_result* of class Clew tells the interpreter that APLAN [APS, 1989] procedure returns a value.
- The procedure should set the result to a predefined name APLAN verdict (*proved*, *refuted*, *not_proved*).
- The input function enters the formulae and description of environmental data types.

In current version of the system we implement interface of *cvc3* prover [CVC3,2012]. However, the number of systems used to test the feasibility can be extended, for example, using the following system: MathSat [MathSat, 2012], Vampire [Vampire, 2012], etc.

Example of programs proving

Let's consider a simple example of an integral function defined recursively: $f(1) = 1$, $f(n+1) = f(n) + (n+1)$.

We write a program to compute this function, using the operators adopted in procedural programming languages:

$fc := 0; k := n;$ /* assignment of initial values */

$L1 : k := k - 1;$ /* the loop calculations */

$L2 : (k \ge 1) \rightarrow (fc := fc + (k+1))else(fc := fc + 1; go\ to\ L3); go\ to\ L1;$ /* verification of the loop */

$L3 : stop;$ /* program termination */

Here n - input variable determines the number of terms calculated amount, fc - output variable contains the result of computing the sum, k - loop variable.

For verification of program by means of IMS system it is required: to annotate a program ("mark" it with annotations) and to prepare verification environment.

Let's name the considering function *Sumpos*. Annotated program of its calculations, prepared for the verification by means of IMS, has the following view:

make_model(
$L0$: *assumption* : $n \geq 1$;
$fc := 0; k := n$;
$L1$: $k := k - 1$;
$L2$: *assertion* : $(Sumpos(n) = fc + Sumpos(k+1) \wedge 0 \leq k \wedge k \leq n)$;
$(k \geq 1) \rightarrow (fc := fc + (k+1))$*else*$(fc := fc + 1;$ *go to* $L3); $*go to* $L1$;
$L3$: *assertion* : $(fc := Sumpos(n))$;
stop);
verify $L0$;

In addition to rewriting rules, the verification environment contains a description of variables of program of function *Sumpos* calculating and formulae used for verification:

(*program environment* : *obj*(*attributes* : *obj*$(fc : \text{int}, k : \text{int}, n : \text{int}, Sumpos : \text{int} \rightarrow \text{int})$;
axioms : $\forall(w : \text{int})($
$(w = 1) \rightarrow (Sumpos(w) = 1) \wedge$
$\wedge (w > 1) \rightarrow (Sumpos(w) = Sumpos(w-1) + w)))$;
initial : $1[empty]$
);

The verification process is accompanied by messages screen output, allowing to monitor system performance. For example, in case of verification program, calculating the function Sumpos, message sequence has the following view:

1) *init*,
2) *start verify* $L0$,
3) *assumption* $(n \geq 1)$ *is consistent*,
4) $fc := 0, k := n$, *go to* $L1$,
5) $k := k - 1$, *go to* $L2$,
6) *assertion* $((Sumpos\ n = fc + Sumpos(k+1)) \wedge 0 \leq k \wedge k \leq n)$ *proved*,
7) *ask*$(k \geq 1), fc := fc + (k+1)$, *go to* $L1$,
8) $k := k - 1$, *go to* $L2$,
9) *assertion* $((Sumpos\ n = fc + Sumpos(k+1)) \wedge 0 \leq k \wedge k \leq n)$ *proved*,
10) *ask*$(\neg(k \geq 1)), fc := fc + 1$, *go to* $L3$,
11) *assertion* $(fc = Sumpos(n))$ *proved*,
12) *stop*, *running model finished*, *all space covered*

Verification was successful.

Let's describe in details the obtained trace step-by-step. *1)* – making initialization of IMS, *2)* – starting model verification from label *L*0, 3) – checking consistency of assumption by using ***Sat*** function, 4) – making assignments and go to the next label *L*1, 5) – make assignment and go to the next label *L*2, 6) – the assertion

was automatically proved, 7) – checking condition, making assignment and go to the next label *L*1, 8) making assignment and go to the next label *L*2, 9) the assertion was automatically proved, 10) the first part of this step is check condition $ask(k \geq 1)$, but this behavior is visited, then checking condition $ask(\neg(k \geq 1))$, making assignment and go to the next label *L*3, 11) the assertion was automatically proved, 12) making end operator *stop*, finish modeling process and this process covered all reachable states of search space.

Conclusion

The developed system has been used successfully in the educational processes of Kyiv Taras Shevchenko National University and Kherson State University in teaching of proving programming. In the future we plan to expand the number of plug-ins in order to prove the satisfiability of formulae, and we will continue to prove partial correctness of programs on new examples (parallel programs, etc.).

Also we hope in the nearest future we will succeed in creation of rest modules, described in section:

"Possibilities: Present and Future".

Bibliography

[IMS, 2011] A.A. Letichevsky, O.A.Letychevskyi, V.S. Peschanenko. Insertion Modeling System //PSI 2011, Lecture Notes in Computer Science, Vol. 7162, Springer, 2011.-p. 262-274.

[APS, 1989] A. A. Letichevsky, J.V. Kapitonova, S.V. Konozenko. Algebraic programming system APS-1. In: O.M.Tammepuu, Informatics'-89, Proc. of the Soviet-Franch symp. Tallin, 1989, p.46-55.

[CYB, 2012] Glushkov Institute of Cybernetics (Ukraine) [http://www.icyb.kiev.ua].

[KSU, 2012] Kherson State University (Ukraine) [http://www.ksu.ks.ua].

[SPEC, 2012] SPEC - Standard Performance Evaluation Corporation[http://www.spec.org].

[Letichevsky, 1996] D.R. Gilbert, A.A. Letichevsky. A universal interpreter for nondeterministic concurrent programming languages//in: M. Gabbrielli (ed.), Fifth Compulog network area meeting on language design and semantic analysis methods, September 1996.

[Letichevsky, 1998] A.A. Letichevsky and D.R. Gilbert. A General Theory of Action Languages// Cybernetics and System Analyses.-1998.-vol. 1.-p. 16-36.

[Letichevsky, 2008] A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, N. Nikitchenko, V. Volkov, and T. Weigert. Insertion modeling in distributed system design// Problems of Programming.-2008.-vol. 4.-p. 13-39.

[Milner, 1989] R. Milner. Communication and Concurrency. Prentice Hall, 1989.

[Milner, 1999] R. Milner. Communicating and Mobile Systems: the Pi Calculus, Cambridge University Press, 1999.

[Hoare, 1985] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[Cardelli, 1998] Cardelli, L. and A.D. Gordon. Mobile Ambients // In Foundations of Software Science and Computational Structures, Maurice Nivat (Ed.).-1998.-LNCS 1378.-p. 140-155.

[Gurevich, 1995] Y. Gurevich. Evolving Algebras // Lipari Guide. E. Borger (ed.), Specification and Validation Methods, Oxford University Press.-1995.-p. 9-36.

[Hoare, 1998] C. A. R. Hoare and He Jifeng. Unifying Theories of Programming. Prentice Hall International Series in Computer Science, 1998.

[Meseguer, 1992] J. Meseguer. Conditional rewriting logic as a unified model of concurrency// Theoretical Computer Science.-1992.- vol. 96.-p. 73-155.

[IMS, 2003] A. Letichevsky, J. Kapitonova, V. Volkov, V.Vyshemirsky, A. Letichevsky Jr. Insertion programming//Cybernetics and System Analyses.-2003.-vol. 1.-p. 19-32.

[APS&IMS, 2012] History of APS&IMS Systems[http://apsystem.org.ua].

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming// Communications of the ACM.-1969.-vol. 12(10).-p. 576-580.

[Floyd, 1967] R.W. Floyd. Assigning meanings to programs// Proceedings of the American Mathematical Society Symposia on Applied Mathematics, 1967, Vol. 19, pp. 19-31.

[SAT, 2012] A. Letichevsky, O. Letichevskyi, T. Weigert, V. Peschanenko. Satisfiability for symbolic verification in VRS // Control Systems and Machines.-2012.-vol. 6 (in print).

[CVC3,2012] CVC3: The CVC3 User's Manual[http://www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html].

[MathSat, 2012] The MathSat 5 SMT Solver[http://mathsat.fbk.eu/].

[Vampire, 2012] Vampire's Home Page[http://www.vprover.org/].

Authors' Information

Alexander Letichevsky *– Head of the Department of Theory of Digital Automatic Machines of Glushkov Institute of Cybernetics. P.O. Box: 40 Glushkova ave., Kyiv, Ukraine, 03187; e-mail: let@cyfra.net*

Major Fields of Scientific Research: automatic-algebraic models of computer systems, algebraic theory of agents and environments, algebraic programming and computer algebra, artificial intelligence, verification

Olexander Letichevskiy *– Researcher of the Department of Theory of Digital Automatic Machines of Glushkov Institute if Cybernetics. P.O. Box: 40 Glushkova ave., Kyiv, Ukraine, 03187; e-mail: lit@iss.org.ua*

Major Fields of Scientific Research: automatic-algebraic models of computer systems, algebraic theory of agents and environments, algebraic programming and computer algebra, artificial intelligence, verification

Marina Morokhovets *- Senior Researcher of the Department of Theory of Digital Automatic Machines of Glushkov Institute if Cybernetics. P.O. Box: 40 Glushkova ave., Kyiv, Ukraine, 03187; e-mail: marina.morokhovets@gmail.com*

Major Fields of Scientific Research: automatic-algebraic models of computer systems, algebraic theory of agents and environments, algebraic programming and computer algebra, artificial intelligence, verification

Vladimir Peschanenko *– Associate Professor of the Department of Informatics of Kherson State University. P.O. Box: 27, 40 rokiv Zhovtnya St., Kherson, Ukraine 73000; e-mail: vpeschanenko@gmail.com*

Major Fields of Scientific Research: insertion modeling, algebraic programming, verification, mathematical pedagogical software, computer algebra algorithms, rewriting.