УДК 519.7

Песчаненко В. С.[1], к.ф-м.н., доцент
Губа А. А.[2], аспірант,
Шушпанов К. І., програміст[3]

Змішаний конкретно-символьний предикатний трансформер

Анотація

Представлена технологія, яка дозволяє у зазначених випадках замість класичного символьного моделювання використовувати пряме виконання C++ коду разом із символьним розв'язанням та доведенням. Описано обмеження на вхідні данні та доведено, що таке розділення задач не призводить до суперечностей, а результат знаходиться у виявлених обмеженнях.

Ключові слова: верифікація, символьне моделювання, інсерційне моделювання, генерація тестів.

[1] Херсонський державний університет, 73000, м. Херсон, вул. 40 Років Жовтня, 27,
e-mail: vladimirius@gmail.com
[2] Інститут кібернетики імені В.М.Глушкова Національної академії наук України, відділ 100, 03680, м. Київ, пр-т. Глушкова 40,
e-mail: antonguba@ukr.net
[3] ООО «Інформаційні програмні системи», 03680, м. Київ, вул. Боженка 15,
e-mail: costa@iss.org.ua

Peschanenko V. S.[1], PhD, associate professor,
Guba A. A.[2], PhD student,
Shushpanov C. I.[3], software developer

Mixed concrete-symbolic predicate transformer

Annotation

The technique that allows the use of direct execution of C++ code together with symbolic solving and proving instead of classical symbolic modeling is presented. The restrictions on incoming data are described and it is proved that defined separation of tasks does not lead to contradictions and corresponds to the detected constraints.

Key Words: verification, symbolic modeling, insertion modeling, test case generation.

[1] Kherson state university, 73000 Ukraine, Kherson, 40 Rokiv Zovtna street, 27,
e-mail: vladimirius@gmail.com
[2] Department 100 of Glushkov Institute of cybernetics NAS Ukraine, 03680, Kyiv, Glushkova ave., 40,
e-mail: antonguba@ukr.net
[3] LLC «Information software systems», 03680, Kyiv, str. Bozhenko 15,
e-mail: costa@iss.org.ua

Статтю представив д.ф.-м.н., проф. Буй Д.Б. (за результатами конференції TAAPSD'2012)

Introduction

Modern software is becoming increasingly complex and needs to be highly reliable. Testing is commonly used for ensuring the reliability of software but it does not guarantee a detection of all errors in the product. Considering that the volume and complexity of requirements of software are increasing, the problem of automatic verification of industrial projects is actual problem. The technique of symbolic verification of requirement specifications of software systems has shown good results in automatic detection of reachability of deadlocks and violation of user-defined properties [1]. In previous works [1-5], symbolic models of systems, which are transition systems with symbolic states represented by formulae of first order logic, were considered. A relation of transitions between the formulae is determined and marked by basic protocols, which are considered as actions performed in the system. A basic protocol is a formula of dynamic logic $\forall x(\alpha(x,r) \rightarrow < P(x,r) > \beta(x,r))$ and describes the local properties of a system in terms of pre- and postconditions α and β. Both are formulae of first order multisorted logic interpreted on a data domain, P is a process, represented by means of an MSC diagram, and describes the reaction of a system triggered by the precon dition, x is a set of typed data variables, and r is a set of environment and agent attributes. The general theory of basic protocols is presented in [2].

At transitions in the space of formulas, a postcondition is considered as an operator. As the

operator transforms one formula to another, in [3] a term "predicate transformer" was used. Thus, to compute transitions between the states of such models, basic protocols are interpreted as predicate transformers: for a given symbolic state of a system and a given basic protocol, the direct predicate transformer generates the next symbolic state as its strongest postcondition and the backward predicate transformer generates the previous symbolic state as its weakest precondition. These concepts have been implemented in VRS (Verification of Requirement Specifications) system [4] and IMS (Insertion Modeling System) system [5].

Splitting formulae into concrete and symbolic parts

There are two main disadvantages of "classic" symbolic modeling [6]: expensive proving and solving processing during the modeling of a system behavior and inability of processing concrete run-time values, for example, when the decision procedure cannot handle the complex mathematical constraints that are generated or when code that uses native or external libraries shall be processed. Therefore, the formulae shall be split into two parts: concrete and symbolic. For this purpose, we split the formulae of initial state and both pre- and postconditions of each basic protocol into two parts.

Data types

Before talking about splitting formulae into symbolic and concrete parts let us see which data types could be supported by our models. User can use the APS [7] data types in the models:

a) simple data types: Boolean, integer, real, enumerated, and symbolic,

b) structured data types: uninterpreted functional symbols, arrays, and lists of simple date types.

Symbolic type is represented by expression in algebra of free terms. Enumerated type is a user-defined one contains symbolic constants (atomic expressions in algebra of free terms). Parameters and returned value of uninterpreted functional symbols are simple data types only. Let denote that if we are talking about functional, then we mean uninterpreted functional symbol. Arrays are considered as uninterpreted functional symbols with bound restrictions for values of parameters.

Concrete basic protocols technique

User requirements for formula splitting into symbolic and concrete parts can be defined by the following statements:

1. User should define a set of concrete attributes in project which always have concrete values in a model.

2. System should analyze this user's defined set of concrete attributes and split initial environment state and basic protocol into two parts: symbolic and concrete.

3. If this splitting is impossible, then it prints error message and finishes its work.

Set of concrete attributes

In general, it is hard to determine automatically which attribute has concrete values in model. From other point of view, user, who has created a model, knows which attributes always have concrete values in it. So, let *c* be a set of attributes, which always has concrete values in a model. This set is defined by user. *c* is called the set of concrete attributes.

Set c should be finite, because otherwise it is hard to understand what should be done for functionals with integer or real parameters. For example, we have next definition of functional: *f: (int)* $\rightarrow$ *int*, initial state: *f(0) = 0*. So, what should we say about formula $(f(i) = 0) \wedge (i \neq 0)$? All of them could have any integer value and the last formula is unsolvable in terms of concrete model, because it defines infinite set of attributes (…, *f(-2), f(-1), f(0), f(1), f(2),* …). Attribute of symbolic type can be handled as concrete if it consists only from concrete attributes, because, in general, it can have infinite set of values. Abstract lists, which have arbitrary number of values in their elements, cannot be used, because it is impossible to implement this with concrete values only. For example, we have a list: *l: list of int*, initial: *l = (Nil;Nil)*, where ";" marks abstract part, and basic protocol $(get\_from\_head(l)>0) \rightarrow <>1$, where *get_from_head* function returns head of the list. So, let's try to apply it to initial: $\exists x(x>0 \wedge l = (x,Nil;Nil))$. It is wrong to change this formula by concrete value of x, because it is just a property and we don't know any concrete value of x. So, attribute *l* can't be a concrete attribute.

Finally, we can define following restrictions for set *c*:

1) Each concrete attribute from *c* should always have one or finite number of concrete values.

2) Symbolic concrete attribute is allowed if it value is constructed from other concrete attributes from *c*.

3) Functional with integer, real or symbolic parameters could not be used as concrete attribute.

4) Abstract list are forbidden in *c*.

Algorithm of splitting environment state formulae and basic protocols conditions into symbolic and concrete parts.

Let *E(r)* be an environment state. Application of a basic protocol to an environment state looks like the following formula:

$$(\exists(x,r)(E(r)\wedge\alpha(x,r))\neq\varnothing)\Rightarrow$$
$$\exists x\ pt(E(r)\wedge\alpha(x,r),\beta(x,r)),$$

where *pt* is a function of predicate transformer [3]. In general, a postcondition of basic protocol can be considered as following conjunction:

$$\beta(x,r)=A(x,r)\wedge L(x,r)\wedge C(x,r)$$

where *A(x,r)* is a conjunction of assignments, *L(x,r)* is a conjunction of list operators *add_to_tail, add_to_head, C(x,r)* is a formula part of postcondition.

Let $E(r/c)\wedge E_c$ be an environment state, where *r/c* is the set of attributes, which have symbolic values, *c* is the set of concrete attributes defined by user, and E_c is a disjunction of conjunctions of concrete values of attributes from c. Any concrete attribute in *c* has corresponded concrete value in all conjunctions in E_c. For example, let

$r=\{x,y,z\}, c=\{y,z\},$

$E(r)=x>0\wedge((y=1)\wedge(z=1)\vee(y=2)\wedge(z=3))$

then

$E(r/c)=E(x)=(x>0),$

$E_c=(y=1)\wedge(z=1)\vee(y=2)\wedge(z=3)$.

User, who creates a specification, knows attributes, which have concrete values in a project. It means that it is always possible to split such environment into concrete and symbolic parts. From that point of view, the pre- and postconditions could be split in concrete and symbolic part too:

$\forall x(\alpha(x,r/c)\wedge\alpha_c\rightarrow<P_c(x,r/c)>\beta(x,r/c)\wedge\beta_c)$

where α_c is a concrete precondition, β_c is a concrete postcondition, $P_c(x,r/c)$ is obtained from $P(x,r)$ after substitution of concrete values from E_c.

Therefore, the restrictions for basic protocols are following:

5) α_c,β_c shouldn't depend on parameters of basic protocol x.

6) If $l\in c$ and l is a concrete list, then list operators *add_to_tail, add_to_head* can contain constant or expression with attributes from *c* only. It is possible to use an expression with attribute from *c* for calculation of its value after substitution of concrete values.

7) $\beta_c=A(c)\wedge L(c)$ can contain assignments and list operators only, because formula *C(c)* of postcondition expresses some property which should be true after application of basic protocol. It means that attributes from *C(c)* should lose their concrete values and should satisfy *C(c)*. In that case the restriction 1) is broken.

Let $E(r/c)\wedge E_c$ be an environment state, $B=\forall x(\alpha(r/c)\wedge\alpha_c\rightarrow<P_c(x,r/c)>\beta(x,r/x)\wedge\beta_c)$

Theorem 1.

$pt(E(r/c)\wedge E_c\wedge\alpha(x,r/c)\wedge\alpha_c,\beta(x,r/c)\wedge\beta_c)=$
$pt(E(r/c)\wedge\alpha(x,r/c),\beta(x,r/c))\wedge pt(E_c\wedge\alpha_c,\beta_c)$

Proving.

pt function builds lists r_{pt},s_{pt},z_{pt} from postcondition $\beta(x,r/c)\wedge\beta_c$ and formula $E(r/c)\wedge E_c\wedge\alpha(x,r/c)\wedge\alpha_c$, where r_{pt} is a set of attributes expressions from left part of assignment of postcondition, s_{pt} is a set of attributes expressions from formula part of postcondition, z_{pt} is a set of other attributes expressions from formula and postcondition. We know that sets of attribute expressions from pairs $E(r/c)\wedge\alpha(x,r/c)$ and $E_c\wedge\alpha_c$, $\beta(x,r/c)$ and β_c are not intersected. It means that we could split each set r_{pt},s_{pt},z_{pt} on two sets $r_{pt}=r_{r/c}\cup r_c, s_{pt}=s_{r/c}\cup s_c, z_{pt}=z_{r/c}\cup r_c$ and $r_{r/c}\cap r_c=\varnothing$, $s_{r/c}\cap s_c=\varnothing$, $z_{r/c}\cap z_c=\varnothing$, because $r/c\cap c=\varnothing$. Let's write formula which is built by *pt* function.

Let $D(r/c)=E(r/c)\wedge\alpha(x,r/c), D_c=E_c\wedge\alpha_c$, so

$q_i=\exists(x_{r/c},y_{r/c},x_c,y_c)(D(x_{r/c},y_{r/c},z_{r/c})\wedge$
$R_i(x_{r/c},y_{r/c},z_{r/c})\wedge L_i(x_{r/c},y_{r/c},z_{r/c})\wedge$
$E_i(x_{r/c},y_{r/c},z_{r/c})\wedge$
$(\vee_j D_c(x_c,y_c,z_c)\wedge$
$R_j(x_c,y_c,z_c)\wedge$
$L_j(x_c,y_c,z_c)\wedge$
$E_j(x_c,y_c,z_c)))\wedge$
$C(r_{r/c},s_{r/c})=$
$\exists(x_{r/c},y_{r/c})(D(x_{r/c},y_{r/c},z_{r/c})\wedge$
$R_i(x_{r/c},y_{r/c},z_{r/c})\wedge L_i(x_{r/c},y_{r/c},z_{r/c})\wedge$

where $x_{r/c},y_{r/c},x_c,y_c$ are variables which are obtained from $r_{r/c},s_{r/c},r_c,s_c$ sets accordingly, pt replaces attribute expressions from $r_{r/c},s_{r/c},r_c,s_c$ on these variables in $D(r/c),D_c$ and some part of

R_i, L_i, E_i, R_j, E_j. L_i is a symbolic list part of postcondition, L_j is a concrete list part of postcondition.

$R_i(x_{r/c}, y_{r/c}, z_{r/c}) =$

$(r_1 = t_1(x_{r/c}, y_{r/c}, z_{r/c})) \wedge (r_2 = t_2(x_{r/c}, y_{r/c}, z_{r/c})) \wedge ...)$

$R_j(x_c, y_c, z_c) = (r_1 = t_1(x_c, y_c, z_c)) \wedge$

$(r_2 = t_2(x_c, y_c, z_c)) \wedge ...)$

R_i (R_j) is obtained from after substitution of variables instead of attribute expressions from $r_{r/c}(r_c)$ in right-sides of assignments.

To describe the construction of $E_i(x_{r/c}, y_{r/c}, z_{r/c})$, we will consider the set *M* of all pairs of functional expressions of the form $(f(u), f(v)), u = (u_1, u_2, ...), v = (v_1, v_2, ...)$, where f(u) is chosen from list $r_{r/c}$, and f(v) – from lists $s_{r/c}, z_{r/c}$. These functional expressions must be equal, if their arguments were equal before application of basic protocol. Similarly for $E_j(x_c, y_c, z_c)$, $C(r_{r/c}, s_{r/c})$ is a formula part of postcondition. $C(r_c, s_c) = 1$, because of restriction 7.

So,

$pt(D(r/c) \wedge D_c, \beta(x, r/c) \wedge \beta_c) =$

$q_1^{r/c} \wedge (\vee_j q_j^c) \vee q_2^{r/c} \wedge (\vee_j q_j^c) ... =$

$= (q_1^{r/c} \vee q_2^{r/c} \vee ...) \wedge (\vee_j q_j^c) =$

$= pt(D(r/c), \beta(x, r/x)) \wedge pt(D_c, \beta_c)$

Theorem is proved.

Let $E(r/c) \wedge E_c$ be an environment state,

$B = \forall x(\alpha(x, r/c) \wedge \alpha_c(c) \rightarrow$

$< P_c(x, r/c) > \beta(x, r/c) \wedge \beta_c(c))$

Lemma 1.

If $E_c \wedge \alpha_c(c)$ satisfy restrictions 1-4 and basic protocol *B* satisfy restrictions 5-7 then new environment state $pt(E_c \wedge \alpha_c(c), \beta_c(c)) \neq 0$ satisfies restriction 1-4.

Proving.

Formula $pt(E_c \wedge \alpha_c(c), \beta_c(c))$ satisfies restrictions 2-4 because of monotony of *pt* function [2] (if $E_c \wedge \alpha_c(c)$ satisfies restrictions 2-4 then $pt(E_c \wedge \alpha_c(c), \beta_c(c))$ satisfies too). If basic protocol *B* satisfies restrictions 5-7 then $pt(E_c \wedge \alpha_c(c), \beta_c(c))$ satisfies the restriction 1, because it's impossible to make such transition that some attribute from *c* would obtain infinite number of concrete value without formulae part of postcondition, parameter of basic protocol and abstract lists.

So, lemma is proved.

This theorem and lemma mean that the application of a basic protocol which precondition, process, and postcondition are split into concrete and symbolic parts could be considered in the following way:

1) $E_c \wedge \alpha_c(c) \neq 0 \Rightarrow pt_c(E_c \wedge \alpha_c(c), \beta_c(c))$,

2) $(pt_c(E_c \wedge \alpha_c(c), \beta_c(c)) \neq 0) \wedge$
$(\exists x(E(r/c) \wedge \alpha(x, r/c)) \neq 0) \Rightarrow$
$\Rightarrow \exists x\ pt((E(x, r/c) \wedge \alpha(x, r/c), \beta(x, r/c))) \wedge$
$pt_c(E_c \wedge \alpha_c(c), \beta_c(c))$

where pt_c is a concrete predicate transformer function that assigns the concrete values to the attributes from *c* set (sometimes it could create a disjunction of concrete values). This function could be easily translated into C++ language for each basic protocol with using of restrictions 1-7.

Implementation

In base language [2], we rely on the following types of functional symbols: integer, real, Boolean, symbolic, and a set of enumerated data types are defined as simple types. Functional symbols of arity 0 correspond to simple attributes; others correspond to the attributes of functional types or functional attributes. For list types, access functions are defined, and lists can change their values by adding or removing elements to (from) head or tail only. Thus, list types exhibit the behavior of queues. Attributes of array type are also possible. In separate C++ file concrete integer attributes could be represented as int attributes, concrete real attributes could be represented as double attributes, concrete Boolean attributes could be represented as bool attributes, concrete attributes of enumerated types could be represented as attributes of enum type with possible values. Functional attributes of aforementioned types could be represented as STL vector attributes of corresponding simple type. Attributes of list type could be represented as STL stack type. Attributes of array type could be represented as C++ array.

For each basic protocol, a separate C++ function shall be created. It should be defined with the help of the following pseudo code:

```
int apply_bp_name(CEnv *env, node_ptr &src, node_ptr &dst, int flag) {
    int f = 0;
    cv_ptr cv = fpl_get_user_data(src);
    if (cv.IsEmpty())
```

```
            return 0;
        if (concrete_precondition) {
            cv_ptr cv_r = cp_env_data(cv);
            concrete_postcondition;
            dst = fpl_mrg_or( fpl_make_user_data(cv_r), dst);
            f = 1;
        }
        return f;
    }
```

where *f* is returned value of status of application of basic protocol (1 – was applied, 0 – was not), *env* is current environment state, *src* is node of user data (concrete attributes), *dst* is node of user data for result (by default it has value 0), *cv* is current user data structure (a possible disjunction is unfolded inside applying of basic protocol function called apply_bp), *cv.IsEmpty()* is checking of pointer, *concrete_precondition* is checking precondition of concrete basic protocol, *concrete_postcondition* is applying postcondition of concrete basic protocol, name in function name is the name of basic protocol, the functions *fpl_make_user_data*, *fpl_get_user_data*, *fpl_mrg_or*, and *cp_env_data* are described below.

The function *fpl_make_user_data* translates user data structure in Clew [7] tree smart pointer node_ptr:

```
    node_ptr fpl_make_user_data(cv_ptr &cv) {
        long_ref_counter_ptr l;
        l.Attach((my_user_data*) *cv);
        return gfpl->make_user_data(l);
    }
```

where *cv* is a smart pointer to the user data, *l* – counter, *l.Attach* is a function for attaching smart pointers to the existent object, gfpl is a global main interpreter of STG (symbolic trace generator) [1], function make_user_data translates smart pointer into tree representation.

Function *get_user_data* translates a tree representation of user data to the smart pointer. It could return empty object if an input node t is not user data node:

```
    cv_ptr fpl_get_user_data(node_ptr &t) {
        if (t->get_mark() != gfpl-> userdatamrk)
            return cv_ptr();
        cv_ptr cv;
        cv.Attach((ConcreteValues*) *gfpl->get_user_data(t));
        return cv;
    }
```

where function *get_mark* returns a mark of APS node [7], *userdatamrk* is a mark for user data representation in the tree, *cv_ptr()* is an empty object, function *get_user_data* returns a smart pointer of *long_ref_counter_ptr*.

Function *fpl_mrg_or* is used to add a new conjunct into result in *concrete_apply_bp* functions:

```
    node_ptr fpl_mrg_or(node_ptr t1,node_ptr &t2) {
        return mrg(gfpl, gfpl->make_or(t1, t2));
    }
```

where function *make_or* makes a disjunction of two nodes and calls default canonizer for disjunction according to default APS function, *mrg* function constructs normal form using ac_local APS name.

The concrete_apply_bp functions should make a copy of current user data before making some changes in it. It is required because few protocols could be applied for one environment state and all next applications of basic protocols should have the corresponded values. The function *cp_env_data* should have the following pseudo code:

```
    cv_ptr cp_env_data(fpl_ptr& fpl, cv_ptr &prev) {
        cv_ptr cv;
        cv.New();
        …
        return cv;
    }
```

where *fpl* is the main interpreter, *prev* is a smart pointer of source user data, *cv* is a smart pointer of copy structure, *cv.New()* allocates memory for an object. The function should return *cv* object.

The pointer for each defined concrete_apply_bp function should be set into internal representation of apply_bp function:

```
    void init_bp_concrete() {
        std::map<std::string, CHashBP>& mp = dataLoader.get_bpMap();
        mp.find("MSC_name")->second.apply_bp_concrete = apply_bp_name;
        …
    }
```

where *CHashBP* is a class of loaded basic protocol (pre- and postcondition, parameters of basic protocols and its types, etc.), *dataLoader* is an object of project loader of apply_bp, mp is a hash of all loaded basic protocols.

If engine tries to apply a basic protocol with initialized concrete_apply_bp function then apply_bp calls first concrete_apply_bp. If concrete_apply_bp is applicable then apply_bp continues an application of the basic protocol with symbolic part of environment state. If concrete_apply_bp is not applicable, then apply_bp returns 0. The common scheme is presented below:

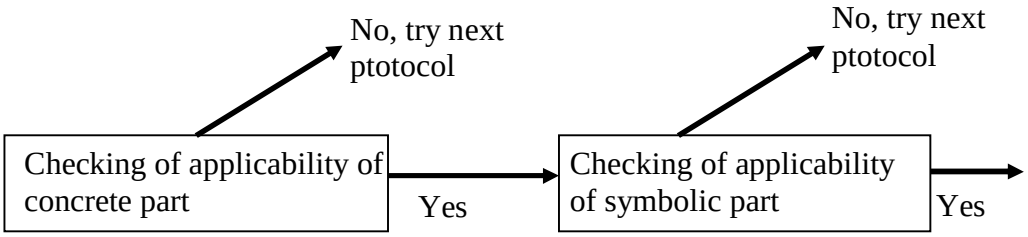

Fig. 1. Common scheme

Experiments

In this section, we present some results from our test suites. All projects were run on symbolic trace generator STG [2] and new mixed concrete-symbolic trace generator STG++.

The project test1 contains functional attribute of symbolic type with integer parameter, simple enumerated and simple integer attributes. All of these attributes initialize with concrete values and have concrete values all time during trace generation (basic protocols do not change those to symbolic ones).

The project test2 contains functional attribute of integer type with integer parameter, several simple enumerated and simple integer attributes, and Boolean attribute. Only functional attributes initialize with concrete values and have concrete values all time during trace generation.

The project test3 contains one functional attribute of integer type with integer parameter, one functional attribute of Boolean type with integer parameter, one functional attribute of symbolic type with two enumerated parameters, and three simple integer attributes. Only two integer and two functional attributes initialize with concrete values and have concrete values all time during trace generation.

Table 1

Comparison of the time on STG and STG++

Project	STG	STG++
test1	2h 30 min	1,8 sec
test2	332,6 sec	0,08 sec
test3	230,1 sec	0,07 sec

Conclusions

Symbolic modeling is a powerful technique for the automated reachability of deadlocks and violations of user-defined properties. We have proposed the technique that helps classical symbolic modeling gain in power in the cases where the proof of the symbolic part can be translated into a direct C++ code. Our technique uses direct execution of code together with symbolic solving and proving.

The nearest plans are investigation of dynamic separation formulae on concrete and symbolic parts, implementation of this technology for messages sending and receiving, timers setting, stopping, and expiration. Using of the technology to reduce interleaving in symbolic modeling and generation of specialized satisfiability and predicate transformer functions for each basic protocol.

References

1. *A. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V. Kotlyarov, T. Weigert.* Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications // Computer Networks. – 2005. – Vol. 47. – P. 662-675.
2. *A. Letichevsky, J. Kapitonova, V. Volkov, A. Letichevsky Jr., S. Baranov, V. Kotlyarov, T. Weigert.* System Specification with Basic Protocols // Cybernetics and System Analyses. – 2005. – Vol. 4. – P. 3-21.
3. *Letichevsky A. A., Godlevsky A. B., Letichevsky A. A. Jr., Potienko S. V., Peschanenko V. S.* Properties of Predicate Transformer of VRS System // Cybernetics and System Analyses. – 2010. – Vol. 4. – P. 3-16.
4. Verification for Requirement Specification (VRS): http://iss.org.ua/ISS_VRS_tool.htm.
5. *A. A. Letichevsky, O. A. Letychevskyi, V. S. Peschanenko.* Insertion Modeling System // PSI 2011, Lecture Notes in Computer Science. – 2011. – Vol. 7162. – P. 262-274.
6. Symbolic modeling, http://en.wikipedia.org/wiki/Model_checking
7. *A. A. Letichevsky, O. A. Letychevskyi, V. S. Peschanenko.* APS and Tools // Bulletin of Kharkiv National University V.N. Karazin. – 2010. – N. 890. – P. 145-153.

Надійшла до редколегії 03.12.2012.