UDC 519.686.2

A. Letichevsky, A. Letichevskyi, T. Weigert, V. Peschanenko

Satisfiability For Symbolic Verification in VRS

Рассмотрены использование логики первого порядка в символьной верификации спецификаций требований программного обеспечения, символьные модели систем, которые есть транзиционными системами с символьными состояниями представленных формулой логики первого порядка. Использованы методы *Satisfiability Modulo Theory* вместо логического вывода в соответствующем исчислении для эффективных вычислений в предикатных трансформерах.

This paper demonstrates the use of the first order logic in symbolic verification of the requirement specifications of reactive software systems. We consider symbolic models of a specified system which are transition systems with symbolic states represented by formulae of the first order logic. To efficiently compute predicate transformers the Satisfiability Modulo Theory methods are used instead of the logical inference in the corresponding calculi.

Розглянуто використання логіки першого порядку у символьній верифікації специфікацій вимог програмного забезпечення, символьні моделі систем, які є транзиційними системами з символьними станами представленими формулою логіки першого порядку. Використано методи *Satisfiability Modulo Theory* замість логічного виводу у відповідних численнях для ефективного обчислення у предикатних трансформерах.

Introduction. This paper demonstrates the use of the first order logic in the verification of requirement specifications of reactive software systems. The presented concepts have been implemented in our VRS (Verification of Requirement Specifications) system [1–3]. The key algorithms used in VRS for the verification of systems are presented. Specifications of software systems in VRS are represented by means of systems of basic protocols. A basic protocol is a formula of dynamic logic $\forall x(\alpha(x,r) \rightarrow \rightarrow <P(x,r)>\beta(x,r))$ and describes the local properties of a system in terms of pre- and postconditions α and β. Both are formulae of the first order multisorted logic interpreted on a data domain, P is a process, represented by means of an MSC diagram [4, 5], and describes the evolution of the specified multi-agent system when triggered by the precondition, x is a list of typed data variables, and r is a list of environment and agent attributes.

Symbolic models of a specified system are transition systems with symbolic states represented by formulae of the first order logic. To compute transitions 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 [6] generates the next symbolic state as its strongest postcondition and the backward predicate transformer [6] generates the previous symbolic state as its weakest precondition. To efficiently compute the predicate transformers we use SMT (Satisfiability Modulo Theory) methods instead of logical inference in the corresponding calculi.

The deductive system of VRS supports the following data types: numeric (integer and real), symbolic (free terms), enumerated, functions (arrays are considered as functions with restricted domains), and queues. This deductive system can be used for static requirement verification of a system by proving that a set of basic protocols specifies a deterministic system and is complete. Predicate transformers can also be used for proving statically the invariance of safety conditions, for dynamic verification of requirement specifications by generating traces of a symbolic system, and for finding deadlocks or safety violations.

In the following section, the specification of a system by means of basic protocols is formalized. We define transition systems explaining the semantics of such specifications for concrete and symbolic models. In section “Base language”, we give an overview of the logic used to express such specifications in our VRS system. In the next section we describe the one of main algorithms VRS relies on during verification: the algorithm for proving satisfiability of formulae, which integrate symbolic and numeric information with behaviors and functional data structures (which makes the solvability of the SAT problem far from obvious).

Section “Verification” describes the use of these algorithms for verification in the VRS system. The conclusion briefly discusses related literature and highlights opportunities for further work.

Basic Protocol Specifications

A basic protocols specification $\sum =< B, E) >$ of a system consists of a set of basic protocols B together with an environment description E. E defines the signature of the language used to express pre- and postconditions of the basic protocols and the interpretation of this signature. We refer to the language defined by the environment description as the *base language* of a given specification; the details of the base language are presented in the next section. E also defines the set of attributes of the specified system (symbols that may take different values over time). Each attribute is of simple or functional type. If an attribute f has functional type $(\tau_1, \tau_2, \ldots) \to \tau$ then *attribute expressions* $f(t_1, t_2, \ldots)$ are available in basic protocols. Further, the types, attributes, and behaviors of agents inserted into an environment as well as the set of possible initial states of agents and the environment are defined in the environment description.

The general form of a basic protocol is

$$\forall x(\alpha(x,r) \to < P(x,r) > \beta(x,r)) . \qquad (1)$$

The variables $x_1, x_2, \ldots$ in the list $x = (x_1, x_2, \ldots)$ are the *parameter* of a protocol and occur under the quantifier together with their types: $\forall x = \forall(x_1 : \tau 1, x_2 : \tau_2, \ldots)$. The list r used in the *precondition* $\alpha(x, r)$, the *process* $P(x, r)$, and the *postcondition* $\beta(x, r)$ of a basic protocol is the list of attribute expressions. The precondition is a formula of the base language, the process is an MSC diagram, and the postcondition is a formula of the base language extended by assignment statements (considered as temporal logic formula).

Each model of a specified system is a transition system with states represented in the form $s[m_1 : u_1, m_2 : u_2, \ldots]$ where s is the *kernel of the environment state*, $m_1, m_2, \ldots$ are the names of agents, and $u_1, u_2, \ldots$ are the behaviors of these agents considered as their states.

All models are the instances of a model of the interaction of agents and environments [7, 8], and a basic protocol specification can be interpreted as a method for defining the insertion function.

The actions of a model are instantiated basic protocols considered atomic (i.e., we do not consider the individual steps in the process of the basic protocol). Depending on how we represent the kernel of the environment state we can distinguish various kinds of models: *Concrete models* represent the state as mappings from the set of attribute expressions to their concrete values. *Symbolic models* represent the state as formulae of the base language. Symbolic models are further differentiated based on the condition under which we consider a basic protocol applicable to a symbolic state $\gamma(r)$. For *universal models*, the applicability condition is the validity of implication $\gamma(r) \to \exists x \, \alpha(x, r)$. For *existential models*, a basic protocol is applicable if $\gamma(r) \wedge \exists x \, \alpha(x, r)$ is satisfiable. In this paper we consider existential symbolic models and their relationship to concrete models.

Concrete models. A concrete model $C_\sum$ defined by the environment description $\sum$ is constructed as follows. A *constant attribute expression* is an attribute of a simple type or an attribute expression of the form $f(a_1, a_2, \ldots)$, where f is an attribute of a functional type and $a_1, a_2, \ldots$ are constant (ground) expressions of corresponding types. Let $A_\sum$ be the set of all constant attribute expressions. The kernel of the state of a concrete model is a partial mapping $s : A_\sum \to D$ from the set of constant attribute expressions to the data domain D (its values). This mapping must preserve the types: if $s(x)$ is defined then the type of $s(x)$ is equal to the type of x. The complete the state of environment $s[m_1 : u_1, m_2 : u_2, \ldots]$ contains all agents inserted into the environment.

The mapping s is extended in a natural way to terms and formulae of a base language through iterative substitution of the values for attribute expressions, and we say that a formula γ is valid on the state s of a concrete model (denoted $s \models \gamma$) if $s(\gamma)$ is defined and valid.

Let $B = \forall x(\alpha(x, y) \to < P(x, y) > \beta(x, y))$ be a basic protocol, $x = x_1, x_2, \ldots$ the parameters of B and $a = a_1, a_2, \ldots$ a list of values such that the type of a_i is contained in the type of parameter x_i (types are partially ordered). A formula $B(a) = = \forall x(\alpha(a, y) \to < P(a, y) > \beta(a, y))$ is called an instantiation of a protocol B. Let $s[m : u] = = s[m_1 : u_1, m_2 : u_2, \ldots]$ and $s'[m'_1 : u'_1, m'_2 : u'_2, \ldots]$ be

the states of a concrete model. Define the transition relation of a system C_Σ in such a way that $s[m:u] \xrightarrow{B(a)} s'[m:u'] \leftrightarrow (s[m:u] \models (m:u) \wedge \wedge\alpha(a,r)) \wedge (s'[m:u'] \models (m:u') \wedge \beta(a,r)])$ where $(m:u) = (m_1:u_1) \wedge (m_2:u_2) \wedge \ldots, (m:u') = (m_1:u_1') \wedge \wedge(m_2:u_2') \wedge \ldots$ An expression $(m_i:u_i)$ asserts that an agent m_i is in a state u_i and is referred to as a *state assertion*.

A transition function defined in this manner may exhibit a high degree of non-determinism. For example, attributes that do not occur in a postcondition may possess arbitrary values and agents that do not occur in the state assumption of a postcondition can change their states arbitrarily. Such nondeterminism can be restricted in a usual way, by restricting that only attribute expressions occurring in postconditions of instantiated basic protocols can change their values and that only agents occurring in such postconditions can change their states (independency constraint). This is possible as in concrete models the values of attribute expressions and the states of agents are independent.

Symbolic models. The kernel of the environment state for an existential symbolic model is a formula of the base language. An environment state has the form $\gamma(r)\,[m:u] = \gamma(r)[m_1:u_1, m_2:u_2, \ldots]$ where $\gamma(r)$ is a formula, and $[m:u]$ defines the states of all agents inserted into the environment. The transition relation must satisfy the following minimal constraints, given $\gamma(r)[m:u] \xrightarrow{B} \gamma'(r)[m:u']$:

- $\gamma(r) \wedge (m:u) \wedge \exists x \alpha(x,r)$ must be satisfiable (*applicability condition*).
- $\gamma'(r) \wedge (m:u') \rightarrow \exists x \beta(x,r)$ (*postcondition requirement*).
- Let $s[m:u] \xrightarrow{B} s'[m:u']$ be an arbitrary transition of a concrete model C_Σ; if $s[m:u] \models \models \gamma(r) \wedge (m:u)$ and $\gamma(r)[m:u] \xrightarrow{B} \gamma'(r)[m:u']$ then $s'[m:u'] \models \gamma'(r) \wedge (m:u')$ (*simulation requirement*).

Under these minimal restrictions, the system again may be highly non-deterministic. More deterministic means of defining the transition relation will be considered below in terms of predicate transformers.

Base language

The underlying language (base language) to capture specifications is a first order multisorted language with interpreted and uninterpreted functional symbols. The interpreted functional symbols characterize the environment and are used for the definition of transition functions, as discussed in the previous section. Uninterpreted symbols are used to represent the changing state of the environment and are identified with attributes of a system. All uninterpreted symbols have types, and thus their possible interpretations are restricted by definite domains and ranges of values. Functional symbols of arity 0 correspond to simple attributes, others correspond to the attributes of functional types or functional attributes.

We rely on the following types (sorts) of functional symbols: *integer*, *real*, *boolean*, *symbolic*, and a set of *enumerated* data types are defined as simple types. For all types the equality predicate is defined. For numeric types the inequality relation is defined, but and only addition and multiplication by constants of the corresponding type are allowed as operations (interpreted functions). Consequentially, arithmetic is limited to linear arithmetic. The domain for a symbolic type is a set of free terms constructed from symbolic and numeric constants by means of a set of predefined constructors. Enumerated data types provide sets of possible values. 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. Behavior types are used as agent states and are considered as elements of behavior algebra [8] (a kind of process algebra). This algebra has two operations: prefixing $a.u$ (a is an action, u is a behavior), and nondeterministic choice $u+v$. It has also three constants: dead lock 0, successful termination Δ and undefined behavior $\perp$. The behavior algebra is generated by constants, actions (arbitrary symbolic constant expressions can be used as actions) and predefined constant behaviors. The last are defined in environment description as minimal solutions of a system of equations in behavior algebra. Functional types are functions from simple types to simple types with arbitrary arity. Arrays are attributes of functional types with arguments (indexes) limited to enumerated types or integers.

All simple types may occur in symbolic data structures. Formulae describing precondition, postcondition and the kernel state of the environment for symbolic models may use only existential quantifiers in positive positions (when there is an even number of negations on any branch leading to the quantifier). Quantifiers can bind only variables of simple types. In preconditions, list data can be used only in access functions. The general form of a precondition is $\sigma(x, r) \wedge F(x, r)$, where $\sigma(x, r) = (m_1 : u_1) \wedge (m_2 : u_2) \wedge \ldots$ is a conjunction of state assumptions and $F(x,r)$ does not contain state assumptions. A postcondition is a conjunction of arbitrary formulae of the base language, state assumptions, and assignments. A simple assignment has the form $x := y$, where x is an attribute expression and y is an expression of a type that is contained in the type of x and asserts that the new value of x is the old value of the expression y. A list assignment adds or removes from the head or tail of a list. The general form of the postcondition is $\sigma(x,r) \wedge R(x,r) \wedge L(x,r) \wedge F(x,r)$ where $\sigma(x,r)$ is a conjunction of state assumptions, $R(x,r)$ is a conjunction of simple assignments, $L(x,r)$ is a conjunction of list updating, and $F(x,r)$ is a formula of the base language without state assumptions.

Assignments are tied to two states: before the application of a basic protocol and after its application, and thus can be considered as temporal logic formulae. Simple assignments can be easily eliminated from basic protocols. For example, the basic protocol $\forall x\, \alpha(x,r) \rightarrow <P(x,r)>(u(v) := w)) \wedge F$ is equivalent to the basic protocol $\forall x\, \forall (y,z)\, \alpha(x,r) \wedge \wedge (y = v) \wedge (z = w) \rightarrow <P(x,r)>(u(y) := z)) \wedge F$. List updating can also be eliminated by a small extension of the base language. Thus, in the previous section we did not consider assignments as part of specification models.

The general form of the kernel state of the environment is $\exists x (L(x) \wedge F(x))$, where $F(x)$ is a formula of the base language without list type expressions and $L(x)$ is a conjunction of equalities of the form $x = y$, where x is a constant attribute expression of a list type and y is a list expression.

Satisfiability

The main functionality of the deductive system of VRS is to check the satisfiability of formulae of the base language. This is used in determining the applicability of basic protocols and in computing predicate transformers.

The applicability of the basic protocol (1) in environment state $\gamma(r)[m : u] = \gamma(r)\,[m_1 : u_1, m_2 : u_2, \ldots]$ is equivalent to the satisfiability of the formula

$$\gamma(r) \wedge (m : u) \wedge \exists x \alpha(x, r)\,. \qquad (2)$$

Checking satisfiability of this formula proceeds in four steps: elimination of state assumptions, elimination of list access functions, elimination of functional attributes, proving closed formula without attribute expressions. These steps are discussed below.

Elimination of state assumptions. Satisfiability of a state assumption given a precondition and a state of the environment is checked by means of matching concrete state assumptions of the environment state and state assumptions depending on parameters and attribute expressions, both considered as matching variables. Matching is performed modulo the following relations: $m : (u + v) = (m : u) \vee \vee (m : v)$, $(m : a.u = n : b.v) \leftrightarrow (m = n) \wedge (a = b) \wedge \wedge (u = v)$. The result will be a new constraint $\gamma'(x, y)$ that replaces state assumptions $\gamma(x, y)$ in precondition and environment state, after which the formula does not depend on state assumptions. Note that there can be multiple results depending on which the state assumptions were matched, where each result gives explicit values for parameters and attribute expressions occurring in the precondition. To prove satisfiability we need only one solution, so these are considered one by one.

Elimination of list access functions. We compute the instantiations of the empty list predicate and the accessor functions. In both cases, a list is matched against $u(x, r)$. The value of the empty list predicate is set to 0 or 1, depending on whether the match succeeded. For accessors, we either obtain a simple type expression a returned from an accessor, or a a new variable v of type τ (the type of the list expression) is generated and added to quantifier prefix (initially empty) of the formula (2). The accessor is set equal to either a or v. The matching of attribute expressions can have several results. Different results are represented as a disjunction of formulae.

First, superpositions of functional expressions are eliminated by the 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 be no more nested functional expressions. For every attribute expression f of array or functional type, all its occurrences $f(x_1)$, $f(x_2)$,… with different parameters x_1, x_2,… are considered: $f(x_i)$ is replaced by variable y_i, bound with an existential quantifier, and equations $(x_i = x_j) \to (y_i = y_j)$ 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,…, then during elimination of functional expression $f(i, u)$, the generated formula will include conjunctive constraints $0 \le i \wedge i \le m-1 \wedge$ $\wedge (u = a_1 \vee u = a_2 \vee \ldots$.

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.

Proving a closed formula without attribute expressions. The deductive system of VRS 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).

The method of proving closed formulae first reduces a formula to prenex normal form and tries to prove it by simple reductions and simplifications. If this is not successful, then the specialized provers are invoked. The first step of the reduction eliminates symbolic constructors. All symbolic equalities of the form $t_1 = t_2$ in which t_1 and t_2 are not variables are analyzed. If, for example, $t_1 = f_1(a_1, a_2)$, $t_2 = f_2(b_1, b_2)$, where f_1, f_2 are symbolic constructors such that $f_1 = f_2$, then this equality is replaced by the equivalent conjunction of equalities $a_1 = b_1 \wedge a_2 = b_2$. For different constructors, $t_1 = t_2$ is replaced with 0. After that all possible substitutions of variables are performed. Given a literal $v = e$, occurrences of variable v are replaced with e, if e does not depend on v and if the type of v includes the type of e. If a symbolic variable v occurs in expression e, the equality is impossible and is replaced by 0. Similarly, 0 is substituted for type mismatches.

Then enumerated types are eliminated. The naive method of replacing a subformula of a form $\exists x P(x)$ where x is a variable of enumerated type with values $a_1, a_2, \ldots$ by the disjunction $P(a_1) \vee$ $\vee P(a_2) \vee \ldots$ is too inefficient due to the large expressions created. Instead, we rely on recursive elimination: after maximally narrowing the scopes of quantifiers, the formulae are proved inside out, expanding the above existential quantification only when a recursive proof was successful.

To prove a closed formula without enumerated types it is reduced to a disjunction of conjunctions of literals bounded with existential quantifiers. It is now sufficient to prove one of these conjunctions. They are transformed to prenex normal form, and a proof search begins. All literals are divided into three groups: equalities, negation of equalities and numerical inequalities. At first, the system of symbolic–numerical equalities is solved. If this system is consistent, then the most general solution $v_1 = e_1 \wedge v_1 = e_2 \wedge \ldots$, where expressions e_1, e_2,… do not depend upon variables v_1, v_2,… is found. Variables v_1, v_2,… are eliminated by a substitution of e_1, e_2,… instead of the corresponding variables in a formula. Now there are only negations of equalities and numerical inequalities. Symbolic negations of equalities $p \ne q$ are eliminated by finding the most general solution of equality $p = q$. If this solution does not exist, then the negation is true and can be deleted. If a general solution $v = t$ for some symbolic variable v exists, the symbolic variables can take on infinitely many values, and therefore it is possible for this variable to take on a value which make equality $v = t$ false. Therefore a literal $p \ne q$ is solvable and can be removed. If a general solution contains numerical equalities only, we add the negation of this solution to the numerical inequalities. After the removal of all solved negations with symbolic variables, there remains a pure numerical formula which is solved by the appropriate numerical prover.

Verification

The input language of VRS expresses requirements specifications for systems through basic protocols. The details of this language have been determined based on a large number of industrial projects aimed at the development of software for distributed reactive systems in various industries. These notations include only some abstractions necessary for the eventual description of algorithms and ignores details connected with the representation of the process part. Neither is efficiency a concern at this point. VRS allows the use of various representations of requirements, from tabular forms to scenario languages. These representations are translated into basic protocols by front-end tools.

VRS is comprised of two groups of tools. The first group is aimed at static requirement checking (SRC), the second is aimed at the dynamic verification of system behavior.

SRC is supported by the following tools: a consistency checker, a completeness checker, a safety checker, and a reachability checker. A basic protocol specification is consistent if under the same state assumptions there is only one basic protocol that can be applied. This condition ensures determinism of a specified system comprised of behaviors of agents that may be nondeterminate. Consistency is checked by proving the satisfiability of the preconditions of arbitrary two protocols. If their conjunction is not satisfiable they are consistent. Completeness is considered to be the validity of the disjunction of preconditions of basic protocols with the same state assumptions and is checked by means of a satisfiability algorithm, i.e., its negation must be unsatisfiable. Completeness is a sufficient condition for the absence of dead locks.

Both consistency and completeness conditions are only sufficient. If a violation has been found we must prove that such state is reachable from the initial states of a system. For this purpose the static reachability checker can be used which tries to prove the safety of the negation of such violation. The reachability checker and the safety checker use the direct predicate transformer defined above. To prove that a given condition is a safety condition, the checker generates the following invariants: if a condition is valid before applying a basic protocol, then it will be valid after its application.

Dynamic verification tools generate traces of a model of the specified system. We provide two trace generators: The concrete trace generator (CTG) works with concrete models of a system and uses a restricted base language limiting assignment statements to the postconditions. The CTG is similar to a model checker and uses heuristics and abstractions to reduce the state space, decrease interleaving, etc. The symbolic trace generator (STG) imposes no restrictions on the base language and uses both direct and backward predicate transformers[6]. The former is used for search for traces from initial to goal states, the latter is used for finding traces from a goal state to initial states. STG and CTG rely on a common generating engine and common tools for controling the search. Both generators can be controled by intermediate goal states and check the validity of given or predefined safety conditions, such as the absence of dead locks in the traversed state space.

Conclusions

We have outlined the key algorithms used in the VRS system for the symbolic verification of requirement specifications. VRS was successfully piloted in a number of industrial projects in a large corporation. The main application domains were telecommunication, automotive, and telematics. Industrial projects successively piloted using VRS consisted of up to 10,000 requirements formalized as basic protocols, with over thousand attributes. Substantial numbers of requirements defects were detected by applying the VRS tools to these industrial specifications. The trace generators have also been leveraged for the generation of tests, and verified specifications have been used as the starting point for further product development.

The theoretical foundation of VRS is the theory of interaction of agents and environments [7, 8], now referred to as insertion modeling [9]. Traditional mathematical models for specifications of concurrent systems usually are based on process algebras (CSP [10], CCS [11], Lotos, ACP, μCRL, π-calculus, etc.), temporal and dynamic logics (LPTL, LTL, CTL, CTL*, PDL), and automata models.

Temporal logic is a formal specification language for the description of behavioral properties of non-terminating and interacting (reactive) systems. Traditional one distinguishes between safety (“something bad never happens”), liveness (“something good will eventually happen”), and various fairness properties. For example, Lamport's TLA (Temporal Logic of Actions) [12, 13] is aimed at the description of such properties and is based on Pnueli's temporal logic [14] with assignment, enriched signatures, and module specifications.

Many temporal logics are decidable and corresponding decision procedures exist for linear and branching time logics [15], propositional modal logic [16], and some variants of CTL* [17]. In such decision procedures techniques from automata theory, semantic tableaux, or binary decision diagrams (BDD) [18] have been used. Typically, a system to be verified

is modeled as a (finite) state transition graph, and the properties are formulated in an appropriate temporal logic. An efficient search procedure is then used to determine whether the state transition graph satisfies the temporal formula or not. This technique was first developed by Clarke and Emerson [19], and by Quielle and Sifakis [20] and extended later by Burch et.al. [21].

In the current version of our VRS system, temporal formulae are represented implicitly and are evaluated by checking algorithms. Enhancements are planned to allow the explicit representation of temporal formulae.

The most closely related verification tools to our approach are the SCR toolset [22] and the Action Language Verifier [23]. Different from these systems, VRS uses MSC [4, 5] as the language for capturing the process part of system requirements. This choice of representation was guided by our experience in applying our tools in industrial projects. Powerful extensions to MSC have been proposed: Live Sequence Charts (LSC [24]), Triggered Message Sequence Charts (TMSC [25]), and Object Message Sequence Charts (OMSC [26]). We are investigating similar extensions to the representation of basic protocols as well as new algorithms for symbolic checking and invariant construction.

1. *Basic* Protocols, Message Sequence Charts, and the Verification of Requirements Specifications / A.Ad. Letichevsky, Ju.V. Kapitonova, A.A. Letichevsky et al. // Computer Networks. – 2005. – **47**. – P. 662–675.
2. *Validation* of Embedded Systems / Ju. Kapitonova, A. Letichevsky, V. Volkov et al. R. Zurawski (Ed.) // The Embedded Systems Handbook. – Miami: CRC Press, 2005.
3. *System* Specification with Basic Protocols / Ju. Kapitonova, A. Letichevsky, V. Volkov et al. // Cybernetics and System Analyses. – 2005. – **4**. – P. 3–21.
4. *International* Telecommunications Union. ITU-T Recommendation Z.120: Message Sequence Charts. Geneva, ITU-T, 2002.
5. *International* Telecommunications Union. Recommendation Z.120 Annex B: Formal semantics of Message Sequence Charts, 1998.
6. *Properties* of Predicate Transformer of VRS System / A.Ad. Letichevsky, A.B. Godlevsky, A.A. Letichevsky et al. // Cybernetics and System Analyses. – 2010. – **4**. – P. 3–16.
7. *Letichevsky A*., *Gilbert D*. A Model for Interaction of Agents and Environments / D. Bert, C. Choppy, P. Moses (Eds.) // Recent Trends in Algebraic Development Techniques. Lecture Notes in Comp. Sci. 1827, Springer, 1999.
8. *Letichevsky A*. Algebra of behavior transformations and its applications / V.B. Kudryavtsev, I.G. Rosenberg (Eds.) // Structural theory of Automata, Semigroups, and Universal Algebra, NATO Science Series II. Mathematics, Physics and Chemistry. – Springer 2005. – **207**. – P. 241–272.
9. *Insertion* modeling in distributed system design / A. Letichevsky, Ju. Kapitonova, V. Kotlyarov et al. // Problems in Programming. – 2008. – **4**. – P. 13–39.
10. *Hoare C.A.R*. Communicating Sequential Processes. Prentice Hall, 1985.
11. *Milner R*. Communication and Concurrency. Prentice Hall, 1989.
12. *Lamport L*. Introduction to TLA. SRC Technical Note 1994-001, 1994.
13. *Lamport L*. The temporal logic of actions // ACM Transactions on Programming Languages and Systems, May 1994. – P. 872–923.
14. *Pnueli A*. The temporal logic of programs // Proc. of the 18th Annual Symposium on the Foundations of Comp. Sci., Nov. 1977. – P. 46–52.
15. *Emerson E*., *Halpern J*. Decision procedures and expressiveness in the temporal logic of branching time // J. of Comp. and System Sci. – 1985. – **30** (1). – P. 1–24.
16. *Fisher M*. *Ladner R*. Propositional modal logic of programs // Proc. 9th ACM Ann. Symp. on Theory of Computing, Boulder, Col., May 1977. – P. 286–294.
17. *Emerson E*. Temporal and modal logic / J. van Leeuwen (Ed.) // Handbook of Theoretical Comp. Sci. – Elsevier, (B). – 1990. – P. 997–1072.
18. *Bryant R*. Graph-based algorithms for Boolean function manipulation // IEEE Transactions on Computers. – 1986. – **35** (8). – P. 677–691,
19. *Clarke E*. *Emerson E*. Synthesis of synchronization skeletons for branching time temporal logic // The Workshop on Logic of Programs, Lecture Notes in Computer Science, Springer Verlag, 1981. – **131**. – P. 128–143.
20. *Quielle J*. *Sifakis J*. Specification and verification of concurrent systems in CESAR // Proc. 5th Intern. Symp. on Programming, 1981. – P. 142–158.
21. *Symbolic* model checking: 10^{20} states and beyond / J. Burch, E. Clarke, K. McMillan et al. // Information and Computation, 1002. – 98 (2). – P. 142–170.
22. *Tools* for constructing requirements specifications: The SCR toolset at the age of ten / C. Heitmeyer, M. Archer, R. Bharadwaj et al. // Computer Systems Science & Engineering,2005. – **1**. – P. 19–35.
23. *Bultan T*., *Yavuz-Kahveci T*. Action language verifier // Proc. of ASE 2001, November 2001. – P. 382–386.
24. *Harel D*., *Marelly R*. Come, Let's Play: Scenario-Based programming Using LSCs and the Play-Engine. Springer 2003.
25. *Sengupta B*., *Cleaveland R*. Triggered Message Sequence Charts // Proceedings of SIGSOFT 2002/FSE–10, ACM Press, 2002. – P. 167–176.
26. *Pattern-Oriented* Software Architecture-A System of Patterns / F. Buschmann, R. Meunier, H. Rohnert et al. // Wiley & Sons. – New York, 1996.

Поступила 22.06.2012

Тел. для справок: +38 095 324-1557, +38 055 246-7070
(Киев, Миссури, Херсон)
E-mail: *vladim@ksu.ks.ua*, *vladimirius@gmail.com*