УДК 004.414.23

S.V. Potiyenko
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
Ukraine, 03680 MSP, c. Kiev, Academician Glushkov ave., 40

Symbolic Modeling of Basic Protocols Systems with Arbitrary Number of Agents

С.В. Потиенко
Институт кибернетики имени В.М. Глушкова НАН Украины, г. Киев
Украина, 03680 МСП, г. Киев, пр. Академика Глушкова, 40

Символьное моделирование систем базовых протоколов с произвольным количеством агентов

С.В. Потієнко
Інститут кібернетики імені В.М. Глушкова НАН України
Україна, 03680 МСП, м. Київ, пр. Академіка Глушкова, 40

Символьне моделювання систем базових протоколів з довільною кількістю агентів

A method of symbolic modeling of formal models is considered in the paper. Object of analysis is a domain of multi-component concurrent systems specified in basic protocols language. A problem of dynamic creation and stopping of agents during state-space exploration is considered. Corresponding algorithm has been suggested as an extension of existing forward and backward predicate transformers. It provides ability to introduce arbitrary number of concurrent processes in verification and test generation.
Key words: symbolic modeling, state-space, concurrent systems.

В статье рассматривается метод символьного моделирования формальных моделей. Объектом анализа является домен многокомпонентных параллельных систем, описанных в языке базовых протоколов. Рассмотрена проблема динамического создания и останова агентов во время обхода пространства состояний. Предложен соответствующий алгоритм как расширение существующих прямого и обратного предикатных трансформеров. Он дает возможность вводить произвольное количество параллельных процессов при верификации и генерации тестов.
Ключевые слова: символьное моделирование, пространство состояний, параллельные системы.

В статті розглянуто метод символьного моделювання формальних моделей. Об'єктом аналізу є домен багатокомпонентних паралельних систем, записаних у мові базових протоколів. Розглянуто проблему динамічного створення та зупинки агентів під час обходу простору станів. Запропоновано відповідний алгоритм як розширення існуючих прямого та зворотного предикатних трансформерів. Він дає можливість вводити довільну кількість паралельних процесів при верифікації та генерації тестів.
Ключові слова: символьне моделювання, простір станів, паралельні системи.

Introduction

This work is done in a scope of a problem of errors detection in multi-component software and hardware systems. Typically, in multi-process software, processes and threads work concurrently, can fork and terminate, use shared memory, send and receive signals. In hardware distributed systems, different components can be switched on and off (or appear and disappear in telecommunication domain) and also communicate using various data channels.

Experience of industrial projects shows that significant defects appear at design and coding stages and could be missed during testing stage. So, development of abstract models with further verification and test generation is an actual task.

We consider models of multi-component systems specified in a basic protocols language [2]. Every model contains environment with agents which work concurrently asynchronously and interact between each other by reading and changing attributes. Agents can be created and stopped dynamically. Concrete state of the environment consists of a set of values of environment attributes, values of attributes of all operating agents and a set of agent names. In symbolic modeling we define symbolic state, or simply state, as a set of concrete states and specify it by formula of first order logic with multisort predicate calculus. Transitions are specified by basic protocols.

Basic protocols system

Basic protocol is defined as a Hoare triple $\forall x(\alpha(r,x) \to < P(r,x) > \beta(r,x))$ [1] and expresses the following fact: if a state of the environment satisfies precondition α then the process P may be performed and the state is modified according to postcondition β (here x – a list of (typed) parameters, r – a list of attribute expressions mentioned below). Pre- and postconditions are first order logic formulas, postcondition can also contain assignment operators and operators for creation and stopping agents. Attribute expression is an attribute name of simple type (enumerated or numeric) or a functional expression $r_1(e_1, e_2, \ldots)$, where r_1 is a functional attribute (uninterpreted function) or an array name, $e_1, e_2, \ldots$ are expressions of corresponding types of arguments.

Environment and agents

As it's described above environment state E consists of first order logic formula D and a set of agent names T. Denote it as a pair:

$$E = \langle T, D \rangle$$

Attributes of the environment and any operating agents can be accessed in formula D but we haven't defined a set of agent names T. Agents are separated by types T_i called agent types. Every agent type T_i is considered as dynamic enumerated type. Initially, its domain contains a set of agent names $e^i_1, e^i_2, \ldots, e^i_n$ which are created in initial state. Domain can be empty if no agents are created. We can refine definition of pair E as:

$$E = \langle T_1 = \{e^1_1, \ldots, e^1_n\} \wedge T_2 = \{e^2_1, \ldots, e^2_m\} \wedge \ldots, D \rangle$$

We will use conjunction operation of environment state with formula F. It's defined as:

$$\langle T, D \rangle \wedge F \Rightarrow \langle T, D \wedge F \rangle$$

Predicate transformers

There is a partial transformation $\mu : S \to S$ on complete set S of states of considered model. A function of state transformation under the action of basic protocol is called forward predicate transformer:

$$E' = pt(E \wedge \alpha, \beta)$$

Here E, E' – environment states before and after execution of basic protocol with precondition α and postcondition β.

$$pt(E \wedge \alpha, \beta) = E'_1 \vee E'_2 \vee \ldots$$

Where E_i' is a new pair describing transformed environment state. Disjunction of E_i' appears as a result of identification of arguments of functional expressions [4, 6].

There is another function of state transformation which restores environment state modified by forward predicate transformer under the action of given basic protocol. This function is called backward predicate transformer [5]:

$$E'' = pt^{-1}(E', \alpha, \beta)$$

Let's consider environment transformations while creating and stopping agents during modeling. Obviously, when an agent is created by some basic protocol a domain of its agent type T_i is extended by new element e^i_{n+1} – generated name of new agent. When agent is stopped all occurrences of its attributes in environment state formula D should be substituted by new bounded variables. But there are two approaches to control agent type domain:

1. The name e^i_{n+1} of stopped agent is removed from domain of its agent type T_i. But it's obscure how to transform formula D from environment state with attributes having this name as a value (like $r = e^i_{n+1}$) and functional expressions with this name occurring in arguments (like $f(e^i_{n+1})$). Should such occurrences be removed from formula D or should it be reckoned as a try to access to values that are out of bounds? Anyway, such approach is unable to save any information about stopped agents.
2. Agent type domains are not changed. Here all occurrences of stopped agent name e^i_{n+1} in formula D stay valid. But there rises another problem of infinite growth of agent type domains.

Taking into account needs of industrial projects we have chosen the second approach where information about all operated agents can be saved. Let's now extend predicate transformers for *create* and *stop* operators.

Operators create and stop

Postcondition $\beta(r, x)$ can contain a number of *create* operators in the following form:

$$r_1 := create\,(T_1, u_1);$$
$$r_1 := create\,(T_2, u_2);$$
$$\ldots$$
$$create\,(T_k, u_k);$$
$$create\,(T_{k+1}, u_{k+1});$$
$$\ldots$$

where r_i are attributes which change their values to newly generated agent names, T_i – agent types, u_i – values of special control flow attribute for each created agent.

Operator *stop*(x) can appear only once in postcondition. It stops agent with name x which is, typically, a parameter of basic protocol.

Initial environment state

Introducing *create* and *stop* operators implies a need of initial state refinement. First, we should generate constraint that, initially, all attributes of agent types can obtain values which are names of initial agents only. This constraint should be saved during performing *create* operators. For each simple attribute a of agent type $T_i = \{e^i_1, \ldots, e^i_n\}$ we should add constraint $((a = e^i_1) \vee \ldots \vee (a = e^i_n))$ to initial formula conjunctively.

Consider uninterpreted functions as attributes in a form of f: (T_i, I) $\to$ T_j. Here I is any simple type excepting agent types. For each such an attribute we add the following constraint:

$$\forall(x:T_i, y:I)(((x=e_1^i)\vee\ldots\vee(x=e_n^i))\to((f(x,y)=e_1^j)\vee\ldots\vee(f(x,y)=e_m^j))).$$

This restriction is evidently extended for functions with arbitrary number of arguments.

Processing create operators in forward predicate transformer

Consider agent type $T_i=\{e_1^i,\ldots,e_n^i\}$ and operator *create*(T_i, u).

After performing this operator we should generate new constraints for uninterpreted functions where new arguments are created. Consider attribute f: (T_i, I) $\to$ T_j (here I is any simple type excepting agent types). New constraint is (denote it as R):

$$R=\forall(y:I)((f(e_{n+1}^i,y)=e_1^j)\vee\ldots\vee(f(e_{n+1}^i,y)=e_m^j)\vee(f(e_{n+1}^i,y)=e_{m+1}^j)\vee\ldots).$$

Here e_{n+1}^i is a new generated name. Notice, that we have added all dynamically created agent names $e_{m+1}^j, e_{m+2}^j,\ldots$ of type T_j to allowed values of functional expressions in the form of $f(e_{n+1}^i, y)$.

Agent type T_i should be extended and constraint R added to formula D:

$$\langle T_i=\{e_1^i,\ldots,e_n^i\}\wedge\ldots,D\rangle\xrightarrow{create(T_i,u)}\langle T_i=\{e_1^i,\ldots,e_n^i,e_{n+1}^i\}\wedge\ldots,D\wedge R\rangle,$$

$$\langle T_i=\{e_1^i,\ldots,e_n^i\}\wedge\ldots,D(r_k)\rangle\xrightarrow{r_k:=create(T_i,u)}$$
$$\langle T_i=\{e_1^i,\ldots,e_n^i,e_{n+1}^i\}\wedge\ldots,\exists yD(y)\wedge(r_k=e_{n+1}^i)\wedge R\rangle.$$

Operators *create* should be processed in the order they appear in the text of postcondition β. It should be done before assignments and formula processing.

If some basic protocol is tried to be applied and it contains parameters of dynamic agent types then all values from domains of these types should be allowed including names of dynamically created agents.

Processing stop operator in forward predicate transformer

Operator *stop*(x) can appear only once in postcondition and means stopping of agent x. All functional expressions which correspond to attributes of this agent should be substituted by new bounded variables in the environment state formula (like attributes changed by assignments or formula in postcondition but without arguments identification [4]). Any other functional expressions stay untouched even if they contain the name of stopped agent in arguments. Agent type T_i of stopped agent is not changed also:

$$\langle T_i=\{e_1^i,\ldots,e_n^i,e_{n+1}^i,\ldots\}\wedge\ldots,D\rangle\xrightarrow{stop(e_{n+1}^i)}\langle T_i=\{e_1^i,\ldots,e_n^i,e_{n+1}^i,\ldots\}\wedge\ldots,D'\rangle.$$

Here D' is the formula D after substitutions mentioned above.

Symbolic modeling and test generation

Generally, forward predicate transformer is used in symbolic modeling to explore state-space of a model. The result of exploration is a set of traces leading to formulated goals. We call this process forward trace generation.

Each trace contains symbolic states of the model which are specified by formulas and should be refined before test generation [7, 8]. In this task of refinement we use backward predicate transformer for reverse passing existing trace from reached goal to initial state. It means that we start modeling from goal state where the number of agents is concrete, their attributes (including control flow) are defined and domains of agent types are known (all names generated in forward mode are presented). Therefore, we suggest to start from algorithm for concrete and defined number of agents on each step of modeling.

Processing *stop* operator in backward predicate transformer

In opposite of forward predicate transformer, operator *stop*(*x*) in backward should create new agent. But it doesn't affect agent types because agent type T_i of stopped agent is not changed in forward. Environment state formula *D* isn't changed also because all attributes of stopped agent (which are substituted with bounded variables in forward) will be restored in further backward trace generation.

$$\langle T_i = \{e_1^i, \ldots, e_n^i, e_{n+1}^i, \ldots\} \wedge \ldots, D\rangle \xrightarrow[stop(e_{n+1}^i)]{} \langle T_i = \{e_1^i, \ldots, e_n^i, e_{n+1}^i, \ldots\} \wedge \ldots, D\rangle .$$

Processing *create* operator in backward predicate transformer in scope of existing trace

Operator *create*(T_i, *u*) extends agent type with created agent and adds restrictions to formula *D* in forward predicate transformer. We need to make reverse actions in backward.

First, created agent should be detected. Its special control flow attribute necessarily should have the value *u*. As we consider backward predicate transformer in scope of existing trace created name can be extracted from this trace. Let it be e_{n+1}^i.

Second, the name e_{n+1}^i of created agent should be removed from domain of agent type T_i.

Third, the formula should be cleaned from obsolete functional expressions. It's done by their substitution with bounded variables (analogously to attributes changed by assignments or formula in postcondition but without arguments identification [4, 5]). Functional expression is reckoned as obsolete if at least one of its arguments is equal to created agent name e_{n+1}^i. All such functions should be found in formula *D* and substituted with bounded variables. We don't care about attributes having concrete value e_{n+1}^i because they can obtain this value by assignments or postcondition formula in protocols applied after agent creation in forward trace generation. So they were changed in that protocols and, in backward trace generation, have been substituted by bounded variables earlier. Denote an obtained formula as *D*″:

$$\langle T_i = \{e_1^i, \ldots, e_n^i, e_{n+1}^i, e_{n+2}^i, \ldots\} \wedge \ldots, D\rangle \xrightarrow[[r_k :=]create(T_i,u):e_{n+1}^i]{} \langle T_i = \{e_1^i, \ldots, e_n^i, e_{n+2}^i, \ldots\} \wedge \ldots, D''\rangle .$$

Backward trace generation with hidden agents

Usage of backward trace generation is not restricted by test generation purposes. It can be used for reachability checking which is done without preceding forward trace generation, i.e. without existing traces and states.

It implies a problem of unknown number of dynamic agents (generated by operator *create*) in a state which backward trace generation starts from. Dynamic agents which are not mentioned in this state are called hidden. The problem lies in basic protocols where hidden agents can operate. Such basic protocols can increase a number of operating agents infinitely because new hidden agent can be instantiated from a parameter each time.

There is a challenge for future work to create algorithms for sensible trace generation with hidden agents. In practical usage, the number of hidden agents is bounded by known concrete value in initial state. In this case the problem of infinite growth of agents is absent and described above algorithms can be used with the only remark: while processing *create* operator in backward predicate transformer, agent to be stopped should be chosen non-deterministically from combined set of operating and hidden agents (above, it was taken from a trace).

Examples

We use forward predicate transformer in the following examples.

Environment	Basic protocol ***bp1***	Basic protocol ***bp2***
Enumerated types: V:{v1,v2,v3}; **Agent types:** T, S; **Attributes:** a:T; **Initial agents:** T:{t1,t2}, S:{s1}; **Initial control flow attributes:** T(t1,idle), T(t2,idle), S(s1,idle);	**Precondition:** S(s1,idle) **Postcondition:** S(s1,idle) ∧ create(T, idle)	**Precondition:** S(s1,idle) ∧ ¬(a = t1) ∧ ¬(a = t2) **Postcondition:** S(s1,end)

Table 1. Example 1.

The question: is *bp2* reachable?

First, we add to initial state the following constraint *R*: (a = t1 ∨ a = t2).

Basic protocol *bp2* cannot be applied in the initial state because of added constraint. After *bp1* application agent type *T* is extended by new generated name *tn*. But the attribute *a* has not been changed and initial constraint *R* remained. Consequently, *bp2* is not applicable again. Despite of any number of *bp1* applications, constraint *R* is always actual.

The answer: *bp2* is not reachable.

Let's modify the example – add some attribute *b* of agent type *T* inside the same agent type *T*:

Agent types: T(b: T), S;

The following constraint R_0 should be added to initial state:

(a = t1 ∨ a = t2) ∧ (t1.b = t1 ∨ t1.b = t2) ∧ (t2.b = t1 ∨ t2.b = t2)

After *bp1* application (create new *T*-agent) this constraint remained and new constraint R_1 should be added for attributes of created agent *tn*:

tn.b = t1 ∨ tn.b = t2 ∨ tn.b = tn

To consider uninterpreted functions let's add global attribute *f*:

Attributes: a:T, f:(V,T)→T;

The following constraint R_0 should be saved in initial state:

∀(x:V) ((f(x,t1) = t1 ∨ f(x,t1) = t2) ∧ (f(x,t2) = t1 ∨ f(x,t2) = t2)) ∧
(a = t1 ∨ a = t2) ∧ (t1.b = t1 ∨ t1.b = t2) ∧ (t2.b = t1 ∨ t2.b = t2),

or it can be written as:

∀(t:T, x:V) ((t = t1 ∨ t = t2) → (f(x,t) = t1 ∨ f(x,t) = t2)) ∧
(a = t1 ∨ a = t2) ∧ (t1.b = t1 ∨ t1.b = t2) ∧ (t2.b = t1 ∨ t2.b = t2).

After *bp1* application (create new *T*-agent with name *tn*) new constraint R_1 should be added:

$$\forall(x:V)\ (f(x,tn) = t1 \vee f(x,tn) = t2 \vee f(x,tn) = tn) \wedge (tn.b = t1 \vee tn.b = t2 \vee tn.b = tn)$$

Now, consider operator *stop*. When some agent is stopped all its attributes should be substituted by bounded variables (like it's done with changed attributes in postcondition but without argument identification).

Environment	Basic protocol ***bp1***	Basic protocol ***bp2***
Enumerated types: V:{v1,v2,v3}; **Agent types:** T, S; **Attributes:** a:T, f:T→V; **Initial agents:** T:{t1,t2}, S:{s1}; **Initial control flow attributes:** T(t1,idle), T(t2,idle), S(s1,idle);	**Precondition:** S(s1,idle) **Postcondition:** S(s1,created) ∧ create(T, idle)	**Precondition:** S(s1,created) ∧ ¬(a = t1) ∧ ¬(a = t2) **Postcondition:** S(s1,end) ∧ f(a) := v3
	Protocol ***bp11***	Protocol ***bps***
	Precondition: S(s1,idle) **Postcondition:** S(s1,created) ∧ a := create(T, idle)	∀(n:T) **Precondition:** T(n,idle) **Postcondition:** T(n,idle) ∧ stop(n)

Table 2. Example 2.

The question is: when is bp2 applicable?

As we consider agent types as dynamic enumerated types we extend them with new elements while performing *create* operators. But we wouldn't remove created elements from agent types after *stop* operators.

The answer is: *bp2* will always be applicable after first creation of *T*-agent (protocols *bp1* or *bp11*). We can save information about stopped agents in attributes of functional types which is useful for users but problematic of searching visited states. After each creation of *T*-agent by protocol *bp1* protocol *bp2* can generate more branches.

Conclusions

Language of basic protocols has been chosen as a formal representation of analyzed models for symbolic modeling. An algorithm for support of dynamic creation and stopping of agents has been developed on a base of existing forward and backward predicate transformers [4, 5, 6]. It provides ability to deal with unknown number of concurrent processes in verification and test generation tasks.

References

1. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. – 1969. – Vol. 12(10). – P. 576–585.
2. A. Letichevsky, J. Kapitonova, V. Volkov, A. Letichevsky Jr., S. Baranov, V. Kotlyarov, T. Weigert. Specification of systems using basic protocols // Cybernetics and System Analysis. – 2005. – N 4. – P. 3–21. (in Russian)

3. Amir Pnueli, Ofer Strichman. Reduced Functional Consistency of Uninterpreted Functions // Electronic Notes in Theoretical Computer Science (ENTCS). – 2006. – Vol. 144. – Issue 2. – P. 53–65.
4. Potiyenko S. Methods of forward and backward symbolic modeling of systems specified by basic protocols // Problems in Programming. – 2008. – № 4. – P. 39–45. (in Russian)
5. Godlevsky A., Potiyenko S. Backward transformation of formulas in symbolic modeling: from the result to the source formula // Problems in Programming. – 2010. – N 2–3. – P. 363–368. (in Russian)
6. Letichevsky A., Godlevsky A., Letychevskyy O.(jr.), Potiyenko S., Peschanenko V. Properties of VRS predicate transformer // Cybernetics and System Analysis. – 2010. – Volume 46. – P. 521–532. (in Russian)
7. A. Kolchin, A. Letichevsky, V. Peschanenko, P. Drobintsev, V. Kotlyarov. An approach to test scenarios concretization in scope of test automation technology of industrial software projects // Modeling and Analysis of Information Systems. Yaroslavskiy National University named after P.G. Demidov. – 2012. – N 6. – P. 79–91.
8. A. Kolchin, V. Kotlyarov, P. Drobintsev. A method of test scenarios generation in insertion modeling environment // Control Systems and Computers. – 2012. – N 6. – P. 43-48.

RESUME

S.P. Potiyenko

Symbolic Modeling of Basic Protocols Systems with Arbitrary Number of Agents

The paper considers a method of symbolic modeling of multi-component concurrent systems specified in basic protocols language. Such systems contain environment with agents which work concurrently asynchronously and interact between each other via shared memory. Agents can be created and stopped dynamically. Symbolic state of the system covers a set of concrete states and is specified by formula of first order logic with multisort predicate calculus. Transitions of the system are specified by basic protocols.

An algorithm for support of dynamic creation and stopping of agents has been developed on a base of existing forward and backward predicate transformers which are functions for symbolic states transformation. It has been specified for verification and test generation purposes. It provides ability to analyze systems with arbitrary number of concurrent processes.

Статья поступила в редакцию 04.04.2013.