Automated creation of verification model for C-programs

Yury Yusupov
Saint-Petersburg State Polytechnic University
Saint-Petersburg, Russia
yury.yusupov@gmail.com

Vsevolod Kotlyarov
Motorola
Saint-Petersburg, Russia
Vsevolod.Kotlyarov@motorola.com

***Abstract*—This paper presents initial results of experiments on deriving of verification models from the source code of a software system. The main principles of C-program transformation into formal view in basic protocols notation as well as realization based on usage of the code analyzer KlocWork are presented. Model adequacy and completeness were checked using VRS technology. Further perspectives and ways of development of the suggested approach are also discussed.**

I. Introduction

Software is often used in areas that are very sensitive to program errors. That's why achieving the highest scores for software quality (interpreted as the number of post-release defects in a software product) is an integral part of the software development life cycle. To ensure the required quality level formal verification (checking that actual properties of a system model meet the requirements [1]) is used along with testing of the final executable code. Also, under certain constraints a formal verification process ensures the completeness of system analysis: all system properties will be considered in all possible modes of the system behavior.

Verification approaches can be divided into two parts: deductive, based on automatic theorem proving using of graphs and different algebras [2], and model checking, based on creating of a program model in some formal representation and checking the system requirements for every possible state of the model [3].

There are many technologies and methods of testing and verification based on the model checking approach – UniTesk[1], Zing[2], Spec Explorer[3], and others ([4, 5]). However, efforts to create formal specifications for any of those approaches are critical for using it in industrial projects, which usually experience heavy time and cost pressure. For example, in UniTesk and Zing the size of formal specifications to be created for a software system is nearly equal to the size of its source code. It is obvious, that creating such specifications is a very laborious task that brings down the attractiveness of program verification with model checking.

This paper outlines the main principles of verification models creation from C-program in the formal language of basic protocols [6] and a realization of this automatic model generation based on the static code analyzer KlocWork [7]. VRS technology [6] was used for approbation of the derived models and its results are presented as well. Also, perspectives and further development issues are discussed.

[1]http://www.unitesk.ru.
[2]http://research.microsoft.com/zing.
[3]http://research.microsoft.com/specexplorer.

II. Scope of the work

A. Basic protocols

The formalism of basic protocols was proposed by the mathematician A. Letichevsky et al. for creating of system behavior models suitable to automated verification [8, 9]. It is based on the theory of agents and environments with the insertion function [10].

Basic protocol is a formal representation of an assertion about some actions that have to be applied in a program or algorithm under some conditions. Consider for example the following requirement for a system: "If a system is in the busy state, it shall be switched to the idle state when the phone goes on hook". This assertion can be divided into three parts: pre-condition – "If a system is in the busy state", process part – "the phone goes on hook", and post-condition – "the system shall be switched to the idle state".

In a general case a basic protocol is a Hoare's triplet [11] in the following notation:

$$\alpha \xrightarrow{\mu} \beta$$

where α and β – are the pre-condition and the post-condition respectively and μ – is the process part of the basic protocol. Both α and β conditions are specified by logic formulas of the basic protocols language (a variant of the first-order logic) which can be evaluated for any state of the system.

Basic protocols can be consistently concatenated through their pre- and post-conditions – if the state specified by the post-condition of one basic protocol is to the same as the one specified by the pre-condition of the next basic protocol (actually, the pre-condition formula of the successor should be derivable from the post-condition formula of the predecessor). All such possible concatenations construct the model's behavior graph to be parsed by verifier.

For this formalization the software system should be interpreted as agents interacting in an environment [10]. An agent has a name and parameters. The set of agent's parameters is specified by the agent's type. At each moment of time when the model is "being executed" by the verifier, each agent

is characterized by the agent's state and current values of its parameters. Agents' interaction is expressed through the messaging mechanism.

The mathematical notation "agent *A* of the type *T* is in the state *s*" can be written as *T(A,s)*. Access to a parameter *p* of the agent *A* of the type *T* has the following syntax: *T A.p*.

The language of basic protocols has an MSC-type syntax [6] and basic protocols can be presented in two ways: textual and graphical (MSC/PR and MSC/GR) [12]. The following elements of the MSC language are used for constructing basic protocols:

- instance – for agents representation;
- condition – for pre- and post-condition formulas;
- action – for description of local actions in a basic protocol;
- message – signal of interaction between agents.

The abovementioned sample requirement translated into a basic protocol is shown in Fig. 1 in a graphical representation.

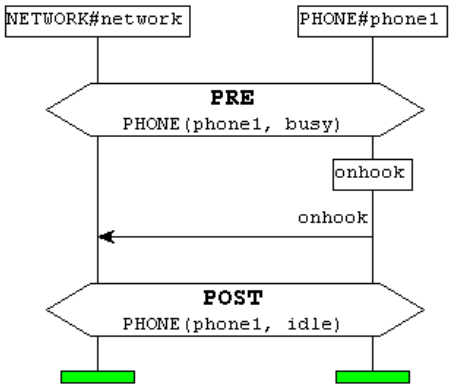

Fig. 1: Basic protocol in a graphic representation

The pre-condition (with the keyword PRE) of the basic protocol is at the top of the diagram, where the state of the model is specified. In this example the model state is defined by the agent state *phone1* which has the value *busy*. The agent *phone1* is of the type *PHONE*.

The process part of a basic protocol describes a set of events that occur in the system after the pre-condition becomes true. This basic protocol contains a local action *onhook* (the phone goes on hook) and sending a respective signal to the *network* agent (which is of the type *NETWORK*) to inform it about the occurred action.

As the result of the events in the process part the system transits to the post-condition state (specified with the keyword POST), where the agent state has changed to *idle*.

B. VRS technology

The VRS technology was developed through a joint effort of the V.M. Glushkov Institute of Cybernetics in Kiev and the St. Petersburg Branch of ZAO "Motorola ZAO" for verification of behavioral specifications of software systems [6]. Using an elaborated combination of model checking and deductive reasoning, the technology is capable to verify models represented with basic protocols, from small to huge ones. As a result, various incidents of non-deterministic behavior, unreachability of specified system states, or deadlocks are detected. If no such defects are found, the system model is formally proved to be complete and consistent within the specified constraints.

Automated verification of software systems with VRS technology implies the functional requirements, which were used for system implementation, and system's model in the form of basic protocols created from the source code, formal specifications, etc. The technology checks that the model meets the system requirements. This means that the software system satisfies them as well.

For verification process an ordered list of signals or basic protocols that contain required events (actions, signals, etc.) should be specified. VRS can check that for this model the behavior graph contains paths which include the specified sequences in the specified order. The existence of such paths (traces) is a proof of correctness of the model behavior with respect to this criterion. Search of such traces is realized by looking for respective signal interaction between agents or by looking for the specified basic protocol names in the generated traces and considering their actual ordering.

Thus, a trace is a scenario of a possible model behavior. Since the model was derived from an actual implementation of a program system, we can say, that a trace is a scenario of an actual system behavior. Scenarios are represented as consistent concatenations of relevant basic protocols into one chain. VRS outputs traces in the MSC/PR view.

Results of verification are automatically summarized in a verification report, which describes all found inconsistencies, discrepancies, deadlocks, and other errors in the model. Traces demonstrating the incorrect model behaviors are attached to the report. They are used to identify the root causes of such incidents.

Traces generated by VRS can be used for automated creation of an exhaustive test suite for the program system. The TAT (Test Automation Toolset) [13] tool is used for automated test generation from those traces along with the respective testing environment and subsequent test runs. In [14, 15] the VRS/TAT system was adjusted for application and system software testing of Java mobile phones.

C. Code analyzer KlocWork

In this work a decision was made to use an existing code analyzer, namely KlocWork, created and maintained by namesake vendor[4]. KlocWork is an industrial analyzer of C, C++, and Java source code, specifically designed for systems of millions of lines of code; it has an open interface to its intermediate code representation (AST – Abstract Syntax Tree), and allows the users to expand its basic functionality.

The main features of the tool are:

1) large amount of code being analyzed (4-5 MLOC);
2) high speed of code analysis due to optimized algorithms of data processing;

[4]http://www.klocwork.com.

3) usage of unified interfaces that allow the users to access intermediate data resulting from the static and dynamic source code analysis.

KlocWork performs syntax analysis of the source code and checks whether it satisfies certain properties, both default and user-defined. The mechanism of analysis is based on various checkers realized as dynamically loaded libraries (dll). Each checker contains rules for AST traversal to check certain source code properties and can be extended by the user in C/C++ [16].

III. Principles of C-code representation in form of basic protocols

A. Main ideas

A C-application in terms of basic protocols can be represented as two agents of different types – agent-application and agent-environment. The agent-application models the whole application and has a set of parameters – all variables of the initial program. Basic protocols are created for every program operator: assignment, conditional operator, loop, etc. A compound statement is represented by several protocols – for the statement as a whole and for all its operands.

The agent-environment models the program environment and is used for exchanging messages with the agent-application. Messages from the agent-application usually inform the agent-environment about events occurring in the application and are used by VRS in verification process for traces search. These messages also provide a useful way for manual analysis of traces.

The pre-condition of a basic protocol describes the application state before execution of the current operator. The state of agent-application is changed in accordance with the current position of the verifier in the model. The values of the agent parameters are changed in accordance with algebraic operations which are executed over these parameters. These operations correspond to algebraic operations over variables of the statement under consideration.

Changes of the agent state are reflected in the basic protocol post-condition, where the application state after execution of the current operator is specified.

The process part of a basic protocol contains a local action with information about the processed statement in its body and interacting signals between the two agents. Signals are divided into two groups: calling functions and accessing parameters.

B. Details

Control flow saving. In order to store a program control flow in the model and to simplify the process of traces generation, a special attribute is added to formulas of the pre- and post-conditions of every basic protocol in addition to the agent state description. It defines the order of basic protocols in accordance with the control flow of the initial C-application. The value of this attribute is changed in the post-conditions of all basic protocols, providing uniqueness to the model states.

Model signals. Two signals with a function name as their parameters are used to indicate a function call: Function_Call_Start(parameter) and Function_Call_End(parameter). The first signal is in the basic protocol, which passes control to a set of basic protocols describing the called function. The second one – in the protocol, which returns control to the main flow from the called function. Besides, the local action of the first protocol contains the called function name with all its parameters.

Two other signals – Variable_Used(parameter) and Variable_Modified(parameter) – are used to indicate access to program variables (using them or changing them respectively). The signal parameter is the variable's name.

Local actions. A local action in a basic protocol is used for better traces readability; its body contains the name of the formalized construct. For example, for a statement with a function call the local action contains the name of the called function with all its parameters.

IV. Realization

The process of C-code formalization and obtaining a program model in form of basic protocols can be divided into two main stages. At the first, KlocWork analyzes the source code and creates its AST. Then the checker parses the tree and generates basic protocols. The respective plug-in was realized in C using special methods of AST traversing, described in [16].

The following data was used for basic protocols creation:

1) model, env – agents, which represent the program system and its environment.
2) MODEL, ENV – the respective model and env agents' types.
3) MODEL#model, ENV#env – instances, which represent agent-application and agent-environment in MSC-diagrams.
4) MODEL(model, FUNC) – representation of the model agent state. FUNC is the name of the function being executed at the current moment.
5) FUNC_VAR – variable VAR of the function FUNC.
6) control_flow – attribute which is used for control flow saving.
7) function_call_point – attribute which is used for saving the function call point.

Module of basic protocol generation used the MSC/PR template:

---

mscdocument *file name***;**
msc *basic protocol name***;**
ENV#env: instance;
MODEL#model: instance;
all: condition PRE /*MODEL(model, *agent state***);**
*atribute and agent parameters****/;**
MODEL#model: action '*local action***';**
MODEL#model: out *signal***(***parameter***) to ENV#env;**
ENV#env: in *signal***(***parameter***) from MODEL#model;**
all: condition POST /*MODEL(model, *agent state***);**
*atribute and agent parameters****/;**
ENV#env: endinstance;

MODEL#model: endinstance;
endmsc;

V. The current status, approbation, possible ways of application and further development

Testing of developed tools was performed on small project pilots (about 1-5 KLOC). Node handling procedures and algorithms of control flow (the order of basic protocols in accordance with the program logic) saving were checked. The experiments demonstrated that creating a model for a project with 350 functions takes about 10 seconds. About 2000 basic protocols were generated during the process.

At the current moment procedures of handling and creating basic protocols for all AST nodes of C-programs have been implemented. Further work will be aimed at deeper analysis of C-constructs, because the complete handling is performed so far only for expressions with unary operations over single operand and binary operations over two operands (assignment). Also, possible ways of links and system functions calls (for example, functions of string handling) formalization for higher model adequacy will be investigated.

Practical applications of the results of this research are as following. As the set of basic protocols contains a formal description of the algorithm realized by the program, the derived model can be used for verification of the model's behavioral properties. Besides, checking of various characteristics of program safety (stack overflow, using of uninitialized variables, etc.) is performed during model execution. The VRS tool allows creating the call graph of a C-application for subsequent model studying and analyzing; also navigation through basic protocols becomes easier with this graph.

It is still too early to expect full automation of formal model creation from C-programs with the described approach, because generated basic protocols still need manual analysis and formatting of their pre- and post-conditions. In some cases pre- and post-conditions need to be supplemented with a formula in the basic protocol language in order to restore the missed program logic, which was bypassed by the checker because of shallow analysis of constructs.

In the near future we plan to apply the described tools and approach to a real Motorola project to reveal defects more thoroughly and select priorities for further development. Besides, further work will be directed to adaptation of this approach to applications in C++ and Java.

VI. Conclusion

Except for engineering development and experiments with the current version of the proposed tool, it is supposed to pursue other research tasks. Using basic protocol generated from the source code of a program for traces generation, we obtain behavioral scenarios which model the program behavior. These scenarios are nothing more than a detailed description of the program behavior, because they describe it at the lower enough level (the source code level) of abstraction. Sometimes such detailed description is redundant while a model of the program's behavior at a higher level of abstraction (e.g., at the design level) is needed. VRS tools which allow creating multilevel behavior models (for example, the call graph, a program model at the component level, etc.) can be used for that. Additional advantage of multilevel models, which are used in the VRS technology, is the possibility of their direct execution and, therefore, obtaining multilevel traces. Higher level behavior scenarios can be used for verification and for concise documenting the given software system as well.

References

[1] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Standard 610.121990, IEEE Standard, IEEE, NY (1990).

[2] Sipma H.B., Uribe T.E., and Manna Z. Deductive model checking. In Proceedings of International Conference on Computer-Aided Verication, 1996.

[3] Edmund M. Clarke and Holger Schlingloff. Model checking. In A. Voronkov, editor, Handbook of Automated Deduction. Elsevier, 2000.

[4] Visser W., Havelund K., Brat G., Park S., and Lerda F. Model checking programs. Automated Software Engineering Journal, 10(2), April 2003.

[5] Fernandez J.-C., Jard C., Jeron Th., and Viho C. Using on-the-fly verification techniques for the generation of test suites. In Proc. 8th Conference on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, New Brunswick, August 1996.

[6] Letichevsky A., Kapitonova J., Letichevsky Jr., A., Volkov V., Baranov S., Weigert T. Basic protocols, message sequence charts, and the verification of requirements specifications, Computer Networks: The International Journal of Computer and Telecommunications Networking, v.49 n.5, p.661-675, 5 December 2005.

[7] Fisher G. The Next Generation of Source Code Analysis, Klocwork, Whitepaper, February, 2008.

[8] Letichevsky A.A., Kapitonova J.V., Volkov V.A., Letichevsky Jr A.A., Baranov S.N., Kotlyarov V.P., Weigert T. System Specification with Basic Protocols // Cybernetics and Systems Analysis, Volume 41, Issue 4 (July 2005), p.479-493.

[9] Baranov S., Jervis C., Kotlyarov V., Letichevsky A., and Weigert T. Leveraging UML to deliver correct telecom applications in UML for Real: Design of Embedded Real-Time Systems by L.Lavagno, G. Martin, and B. Selic (editors), pp. 323342, Kluwer Academic Publishers, 2003.

[10] Letichevsky A.A., Kapitonova J.V., Volkov V.A., Vyshemirskii V.V., Letichevsky Jr. A.A. Insertion Programming // Cybernetics and Systems Analysis, Volume 39, Issue 1 (January 2003), p.16-26.

[11] Hoare C.A.R. Communicating sequential processes, Prentice Hall, London, 1985.

[12] ITU Recommendation Z.120. Message Sequence Charts (MSC), 11/99.

[13] Drobintsev P.D. Integrated technology of software products quality guaranteeing by means of verification and testing. PhD, SPbSPU. 2006. 238 p.

[14] Karpov A.N. Technology of adjustable tests generation from formal specifications for embedded software and program interfaces realized in Java-similar languages. PhD, SPbSPU. 2007. 145 p.

[15] Golubev A.A. Methods of agents creating and integrating into application and system saftware for testing automation and monitoring of computing embedded systems. PhD, SPbSPU. 2007. 150 p.

[16] Klocwork Extensibility Interface User's Guide, Document version 1.3, KlocWork, 2006.