IEICE TRANSACTIONS on Information and Systems

VOL. E101-D NO. 12
DECEMBER 2018

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

Visualization of Inter-Module Dataflow through Global Variables for Source Code Review

Naoto ISHIDA[†a)], ***Nonmember***, **Takashi ISHIO**[††], ***Member***, **Yuta NAKAMURA**[†††], **Shinji KAWAGUCHI**[†††], **Tetsuya KANDA**[†], ***Nonmembers***, ***and*** **Katsuro INOUE**[†], ***Fellow***

SUMMARY Defects in spacecraft software may result in loss of life and serious economic damage. To avoid such consequences, the software development process incorporates code review activity. A code review conducted by a third-party organization independently of a software development team can effectively identify defects in software. However, such review activity is difficult for third-party reviewers, because they need to understand the entire structure of the code within a limited time and without prior knowledge. In this study, we propose a tool to visualize inter-module dataflow for source code of spacecraft software systems. To evaluate the method, an autonomous rover control program was reviewed using this visualization. While the tool does not decreases the time required for a code review, the reviewers considered the visualization to be effective for reviewing code.

key words: *static analysis, dataflow analysis, software visualization, code review, independent verification and validation*

1. Introduction

Defects in spacecraft software may lead to loss of life and serious economic damage. For example, the Ariane 5 launch failure in June 1996 was the result of a runtime error in its software system [1]. In 1999, the Mars Climate Orbiter was lost because of an incorrect unit conversion in the software system [2]. Both failures resulted in serious economic losses.

To improve the reliability of a spacecraft's software system, a third-party organization reviews the design and source code of the system independently of its development team. This procedure is known as independent verification and validation (IV&V) [3]. The technical independence of IV&V allows a system to be evaluated from an objective view point. This is a key factor in making a system safer [4]. However, a code review in IV&V is generally difficult, because reviewers must investigate a set of source files without prior knowledge. Moreover, the budget and time available for IV&V are limited, and hence it is important to perform an effective code review [4].

Manuscript received May 17, 2018.
Manuscript revised August 2, 2018.
Manuscript publicized September 26, 2018.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with the Graduate School of Science and Technology, Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
[†††]The authors are with the IV&V Research Laboratory, Japan Manned Space Systems Corporation, Tokyo, 100–0004 Japan.
a) E-mail: n-isida@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2018EDL8104

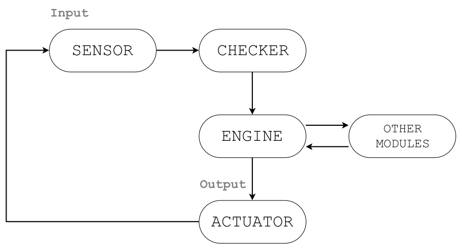

Fig. 1 An example of a control-flow cycle of modules of a spacecraft software

A typical spacecraft software system consists of a highly modularized C program. A module comprising a number of functions plays a particular role in a spacecraft's system. Figure 1 presents an example of a control-flow cycle of modules. The cycle starts with the sensor module, which obtains values from hardware sensors. Then, the checker module removes sensor noises. Next, the engine module computes the state of the spacecraft using the data received from the checker module. Finally, the actuator module controls the hardware based on the recognized state.

Although the design is well organized, the implementation is not straightforward. In order to avoid runtime errors, the system allocates memory statically. In other words, all variables are global and we assume that all global variables are declared extern in a common header file (e.g. global.h). Every module can read and write arbitrary global variables, while local variables and function parameters using a stack are prohibited. Rather than a parameter of a function call, a caller function stores a value as a global variable, and a called function then reads that global variable. Because such implicit dataflow paths are crucial to determining the behavior of a program, reviewers must manually investigate the global variables and then verify whether the dataflow paths are complete.

The goal of this study is to enable reviewers to investigate possible dataflow paths efficiently and exhaustively. To achieve this goal, we propose a tool to visualize dataflow paths among modules in a spacecraft software system. This tool utilizes a reflexion model [5]. While all functions in a system interact with one another using global variables, we only visualize inter-module dataflow paths, so that reviewers can easily compare their expected dataflow paths with the actual dataflow paths.

To evaluate the tool, we performed an experiment using code review tasks with eight subjects engaged in IV&V. The subjects recognized the tool as being effective, although it did not decrease the time required for tasks.

Section 2 describes the proposed tool, and Sect. 3 presents the results of the evaluation experiment. Finally, the paper concludes with a summary and discussion of future work.

2. Visualization Tool

In this work, we propose a tool to visualize inter-module dataflow for source code of spacecraft (and satellite) software systems. The tool supposes that the source code is written in C using only statically allocated variables. The tool also supposes that the development team provides a high-level design model of the system, i.e. how modules such as sensor, controller, and actuator interact to one another, and how the modules are corresponding to functions in the source code. In the beginning of a development process, the development team designs a system as a set of modules and their ideal relationships which the system should follow. Since a module is often represented as a number of functions in actual source code, our tool extracts actual data-flow relationships from functions and translates them into a module level so that reviewers can compare the actual and ideal relationships of modules and find their inconsistencies.

Our tool requires three inputs: the source code to be analyzed, the mapping of modules and functions, and a set of global variables of interest to reviewers (target variables). The mapping provides a list of module names and how each module is corresponding to functions in actual source code. Then, the tool extracts and visualizes the dataflow paths related to the target variables.

The tool generates a directed graph, where each vertex represents a module and each edge represents a dataflow relationship between modules. Figure 2 presents an example of such a graph. A label attached to an edge represents a target variable name related to the dataflow. The tool employs two types of edges: solid and dotted.

- A solid edge from a module M_1 to another module M_2 exists with respect to a target variable g if M_1 writes a value to g and M_2 reads the value of g.
- A dotted edge from M_1 to M_2 exists if M_1 writes a value to g and M_2 does not read g, although M_2 can read the value in g written by M_1.

No edges exist between M_1 and M_2 if the modules do not interact with one another at all using target variables.

In Fig. 2, the solid edge from the SENSOR module to the CONTROLLER module indicates that SENSOR passes data to CONTROLLER using the variable `g_sensor_input.value`. The dotted edge from SENSOR to ACTUATOR indicates that ACTUATOR does not read the value of `g_sensor_input.value` defined by SENSOR, although the value reaches ACTUATOR executed after SENSOR. The graph shows that this variable is only used for an interaction between SENSOR and CONTROLLER. The graph also shows that the additional variable `g_actuator_output.power` is only used for an interaction between CONTROLLER and ACTUATOR.

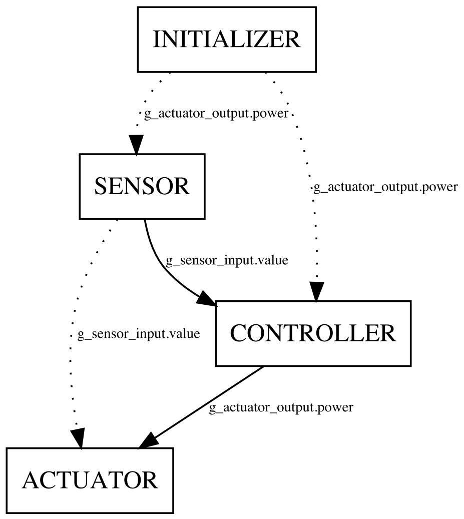

Fig. 2 An example of inter-module dataflow diagram generated by the tool

The implementation of the tool employs two existing OSS components: srcML and srcSlice. srcML [6] translates source code into an XML-based representation. We employed srcML 0.9.5, which is available on the website[†]. srcSlice [7] takes srcML's output as input, and extracts dataflow relationships for each variable in a program. Because srcSlice provides line numbers where each variable is defined and used, our tool extracts the dataflow information for the target variables and translates it into solid and dotted edges between modules. It should be noted that the version of srcSlice that is available on the website[††] does not support global variables, and so we customized the implementation for our tool. This customized version is available on GitHub[†††].

3. Evaluation

In order to evaluate the effectiveness of our visualization, we conducted an experiment consisting of code review tasks using the visualization.

3.1 Procedure

3.1.1 Subjects

We selected eight people engaged in spacecraft software IV&V as the subjects. We divided them into two groups of four people, Group A and Group B, such that on average the groups members have almost the same work experience.

[†]http://www.srcml.org/
[††]http://www.srcml.org/tools.html
[†††]https://github.com/MaxfieldWalker/srcslice-fork

The subjects in Group A conduct code review tasks in their usual environment, and the subjects in Group B perform the same tasks using our visualization.

3.1.2 Target Program

We utilized an autonomous rover control program that runs on Mindstorms EV3 [8] as a code review target. This system has been developed for educational purposes, but is implemented in the manner of real spacecraft software. Functions are clearly separated for each role, in order to achieve a high modularity. However, the system includes many global variables. The system comprises 11 C source files, and the total size is about 500 lines of code.

The program first initializes the state of the system, and then runs a control cycle including three steps: fetching sensor values, calculating the output power based on the sensor values, and driving the motors. Each step is represented by a single function, corresponding to the modules: SENSOR, CONTROLLER, and ACTUATOR, respectively. We selected the structure variables `g_in` and `g_out`, which store the sensor values and motor output, respectively, as target variables.

3.1.3 Code Review Tasks

We asked the subjects the following questions as review tasks.

Q1 Choose the correct roles of the structure variables `g_in` and `g_out` in the program. (Three options are provided with the question.)

Q2.1 The member values of `g_in` should be set only within the SENSOER module. Answer whether or not an exceptional execution path exists where another module overwrites the values.

Q2.2 The member values of `g_out` should only be set within the CONTROLLER module, mainly based on the values of `g_in`. Answer whether or not an exceptional execution path exists where a value of `g_out` is defined without depending on `g_in`.

Q3 Draw dataflow paths among the three modules for the variables `g_in` and `g_out`.

Q1 comprises a warm-up question for the subjects to get used to the target program. The questions Q2.1 and Q2.2 comprise the main tasks. We only provided a visualization result for the target program (Fig. 3) to Group B, so that we could compare the results with those of Group A, who did not have the diagram. The question Q3 aims to evaluate the accuracy of the visualization result and the cost of drawing a similar figure manually. We asked Q3 to Group A only.

We verified the correctness of each answer and measured the times required for Q2 and Q3. We excluded Q1 from the evaluation criteria, because Q1 represents a warm-up question.

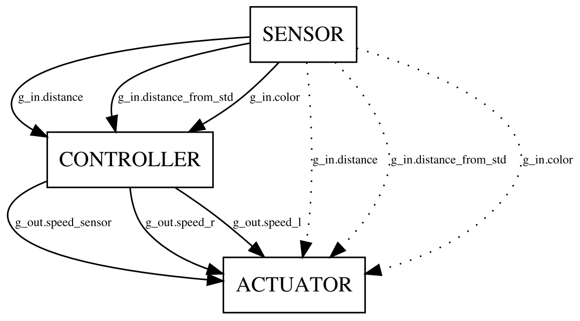

Fig. 3 The inter-module dataflow diagram for the target program

Table 1 The times required to answer the questions (Unit minute)

	Q1	Q2.1	Q2.2	Q3	Sum of Q2
Whole avg.	9.4	18.3	21.5	21.0	39.8
Group A avg. (w/o the diagram)	11.5	11.8	17.8	21.0	29.5
Group B avg. (w/ the diagram)	7.3	24.8	25.3	N/A	50.0

3.1.4 Questionnaire

Following the completion of the tasks, we conducted a questionnaire with the subjects to evaluate the effectiveness of the visualization. We asked the following questions.

- Is the inter-module dataflow diagram effective for code comprehension? (Five-point scale ranging from "Strongly Disagree" to "Strongly Agree") Why do you think so? (Free writing)
- Is it easy to understand what the inter-module dataflow diagram shows? (Five-point scale)
- What is needed to improve the inter-module dataflow diagram? (Free writing)

In the questionnaire, we provided the visualization result to all the subjects.

3.2 Results and Analysis

3.2.1 Code Review Tasks

All of the subjects correctly answered the code review questions. Table 1 presents the times required to answer the questions. For Q2.1 and Q2.2, the average answering time for Group B, working with the visualization, was longer than that of Group A. In the experiment, the number of participants is limited. Since individual ability of participants may vary, it is possible that we were unable to perfectly balance the difference in ability between the two groups. Alternatively, the visualization may have led the participants in Group B to read the source code in more detail because the visualization provides an exhaustive check list of dataflow paths that should be reviewed. Since Group A has no such a list, the participants in Group A could miss some of the data-flow paths. It should be noted that we performed

Table 2 The questionnaire results

	How effective for code comprehension? (1∼5)	How easy to understand? (1∼5)
Whole avg.	4.13	3.88
Group A avg. (w/ the diagram)	4.25	4.00
Group B avg. (w/o the diagram)	4.00	3.75

the Wilcoxon rank-sum test (2-sided) and the time difference is not statistically significant at five percent level (p-value=0.2265).

The average answer time for Q3 was 21 minutes. Although the subjects in Group A read the source code repeatedly to answer Q1 and Q2, they required a considerable time to draw a dataflow graph manually. In addition, dataflow edges are missing in the graphs created by three out of the four subjects. From this result, we can confirm that a manual dataflow inspection is time-consuming and error-prone.

3.2.2 Questionnaire

Table 2 presents the questionnaire results. The average score for the question asking how effective the inter-module dataflow diagram is for code comprehension was 4.13. The average score for the question asking how easy it is to understand the content of the inter-module dataflow diagram was 3.88.

The reasons why the subjects thought the inter-module dataflow diagram is effective for code comprehension are presented below.

- The diagram helps to understand the essential processing flow, ignoring indirectly related variables such as `g_status` (the variable to store the state of the program).
- The diagram visually shows how a value influences others between modules and helps with clear thoughts.
- Using the diagram, it becomes easier to find a starting point for checking how variables are used and in what condition variables are set.

The answers to the question concerning what is needed to improve the inter-module dataflow diagram are presented below.

- The difference between directed edges represented by solid lines and dotted lines is not clear.
- When the size of the source code increases, the diagram will be difficult to see because of the increase in the number of dotted line directed edges. Dotted line directed edges may be useless for a large software project.

In the questionnaire, many favorable answers were reported in answer to why the inter-module dataflow diagram is effective for code comprehension, such as that the diagram helps to understand the essential processing flow. From this, it can be confirmed that the inter-module dataflow diagram helps with code comprehension for people engaged in IV&V.

As a proposal for improvement, participants stated that the difference between directed edges represented by solid lines and dotted lines is not clear, and that it would be difficult to understand the diagram because of too many dotted line direct edges as the program becomes larger. Thus, we should refine the tool by adding the option to switch to showing dotted line directed edges.

4. Conclusion

In this study, we developed a tool that helps to verify dataflow paths for the safety of spacecraft software, and performed an evaluation experiment involving participants who are engaged in IV&V.

In the experiment, we obtained many favorable answers from the subjects engaged in IV&V. The results demonstrate the potential effectiveness of the visualization method. In future work, we would like to make the tool more practical by considering the opinions reported in the evaluation experiment.

References

[1] J.L. Lions, "Ariane 5 flight 501 failure," Technical report, European Space Agency. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, 1996.

[2] A.G. Stephenson, D.R. Mulville, F.H. Bauer, G.A. Dukeman, P. Norvig, L.S. LaPiana, P.J. Rutledge, D. Folta, and R. Sackheim, "Mars climate orbiter mishap investigation board phase I report," NASA, Washington, DC, 1999.

[3] IEEE Standard for Software Verfication and Validation, IEEE Std. 1012-2004, IEEE Standard Association, 2005.

[4] R. Ujiie, M. Katahira, Y. Miyamoto, H. Nakao, and N. Hoshino, "Measurement of JAXA's IV&V activity effectiveness based on findings," Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA), pp.291–296, IEEE, 2011.

[5] G.C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," ACM SIGSOFT Software Engineering Notes, vol.20, no.4, pp.18–28, 1995.

[6] J.I. Maletic, M.L. Collard, and A. Marcus, "Source code files as structured documents," Proc. 10th International Workshop on Program Comprehension, pp.289–292, IEEE, 2002.

[7] H.W. Alomari, M.L. Collard, J.I. Maletic, N. Alhindawi, and O. Meqdadi, "srcSlice: Very efficient and scalable forward static slicing," Journal of Software: Evolution and Process, vol.26, no.11, pp.931–961, 2014.

[8] "31313 mindstorms ev3." https://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313.