applied sciences

MDPI

Article

Improvement Practices in the Performance of a CPS Multiple-Joint Robotics Simulator

Bo Kyung Park [1], Byungkook Jeon [2,*] and R. Young Chul Kim [1,*]

[1] SE Laboratory, Department of Software and Communication Engineering, Hongik University, Sejong 30016, Korea; park@selab.hongik.ac.kr
[2] Department of Software, Gangneung-Wonju National University, Wonju 26403, Korea
* Correspondence: jeonbk@gwnu.ac.kr (B.J.); bob@hongik.ac.kr (R.Y.C.K.); Tel.: +82-33-760-8003 (B.J.); +82-44-860-2477 (R.Y.C.K.)

Received: 29 November 2019; Accepted: 20 December 2019; Published: 25 December 2019

Abstract: In the Fourth Industrial Revolution environment, plenty of automatic and smart software in diverse fields will come out, such as virtual reality/augmented reality (VR/AR), autonomous robots and vehicles, smart factoring, and so on. In Korea, especially, one important issue is the cyber–physical system (CPS), used for monitoring and controlling the smart and automatic system in a smart city. This kind of system, therefore, needs to have good performance; otherwise, it may not respond in time. To solve this, we propose a code visualization approach to reduce code complexity based on a static analysis, which identifies bad codes against performance. To ensure better performance, we make our queries identify performance degradation factors, store statically analyzed data into database (DB) tables, and visualize bad operation patterns. For performance improvement, we can refactor with them. As a result, we reduce the code complexity of CPS-based software to obtain good performance. With this approach, we expect to have better performance and a reduction in the complexity of CPS software without even power consumption.

Keywords: cyber–physical system (CPS); performance measurement; code visualization; refactoring; SW quality improvement

1. Introduction

A cyber–physical system (CPS) handles a huge amount of data from both cyber and physical environments. It is a dependable system that controls real-world systems and processes [1,2]. In other words, a CPS involves interaction with both computers and the real world. A CPS may include a network environment with optimal performance in a given system. A CPS monitors the state of a physical system with various embedded devices, and further performs certain desired operations based on the monitoring information. The CPS's operations affect the physical system, which is complicated by the limitations of computing technology [3]. That is, current embedded systems have been made considerably more complicated, and interoperable through the network. CPS technology may combine information technology (IT) with traditional industries, such as defense, transportation, energy, and automobiles.

A CPS deals with an abstract model, which can easily handle complex systems. However, the model's performance must be validated [4,5]. To date, most testers just verify them with software testing. The CPS model needs to verify software with simulation technology.

The problem with this kind of system is that it should deal with the complexity of various complex systems. Consequently, it cannot help but increase the code size and complexity of CPS-based software. Furthermore, there is no guarantee that the CPS software will have good performance or quality. To solve this problem, we propose to visualize performance degradation factors of the CPS-based

C/C++ code with our visualization mechanism. For performance and quality improvement, we apply our proposed method to the CPS software with the most complex modules, and guild, to refactor them and improve CPS-based C/C++ code.

In this paper, Section 2 discusses related works. Section 3 describes how bad performance factors within CPS-based C/C++ core code are visualized with our visualization approach. Section 4 presents the results before and after refactoring. Finally, Section 5 presents our conclusions and describes further research.

2. Related Works

2.1. Modeling and Software Verification Technology

Techniques such as Modeling and Simulation (M&S) and software (SW) validation are used to measure CPS–SW performance for simplification and to ensure that the system is reliable. In particular, we consider the modeling, that is, a computational model to represent the system before developing it. This is divided into a discrete and a continuous model. The discrete model models a digitized software system, whereas the continuous model models an analog system. In the recently emerged information technology (IT) convergence industry, the use of a hybrid model has been proposed to fuse the discrete and continuous models [2].

Unlike the hybrid model, the domain-specific modeling method reflects the characteristics of the domain. This method performs modeling based on a platform/architecture according to domain-specific characteristics. Particularly, instead of detailed models, some scientists use abstraction techniques to reduce system complexity.

Software validation techniques must validate the source code based on requirement specification. In our cases, we considered checking the performance and quality of the source code with static and dynamic analyses.

2.2. Software Performance

Software performance may mean the measurement of the central processing unit (CPU)'s performance and memory usage. Hardware performance includes parameters such as processing speed, channel capacity, latency, bandwidth, throughput, and power consumption. Software performance includes parameters such as availability, code size, hardware weight, and response time [6,7]. Availability is the probability of the software functions working according to the requirements at the given time. Availability is calculated as follows: the mean time to repair (MTTR) is subtracted from the mean time between failures (MTBF). The obtained value is divided by the average recovery time (MTBF) and is further multiplied by 100.

Most embedded systems have code size and hardware weight limitations. The code size with complexity can have a significant impact on performance. Response time is the total amount of time it takes to respond to a request for service execution. The longer the response time is, the slower the system is [7–10].

In this study, we measure response time for software performance. We extract code degradation factors through our code visualization. Finally, we compare the results before and after refactoring on a procedural language, C/C++ code.

3. Performance Measurement Mechanism for CPS-Based C/C++ Codes

Through our previous code visualization, we identify the code operations of the CPS-based C/C++ code. Then, we measure the performance of core operations, which guide the improvement of the performance for refactoring.

Figure 1 shows how to apply the performance improvement mechanism to procedural language by adapting our existing code visualization [11,12]. We describe the performance measurement and improvement processes as follows: (1) Input the target source code into the performance measurement system, and extract the necessary results through code visualization. This code visualization approach possibly represents the complexity of the code's inner structure. We can identify high-complexity modules of the analyzed results. By adapting this method, we can identify bad performance factors to improve the code's performance, by inputting the entire procedural code. However, for efficient performance, we can guide the refactoring of high-complexity modules for better performance. The selected modules are corrected for refactoring. In the refactoring process, (2) we automatically extract a pattern of performance degradation factors through RuleChecker. The extracted data is stored in SQLite as an extensible markup language (XML) file. Then, (3) we use Visual Studio Performance Testing (VsPerf) for the dynamic analysis of C/C++ code. The data extracted from VsPerf is refined into XML data using VsPerf Data Extractor, and then the extracted data are stored in SQLite. Finally, we (4) extract performance visualization graphs that visualize information, such as C/C++ code data in violation of performance requirements within the existing CPS–SW structural information (coupling) and memory usage information in the module.

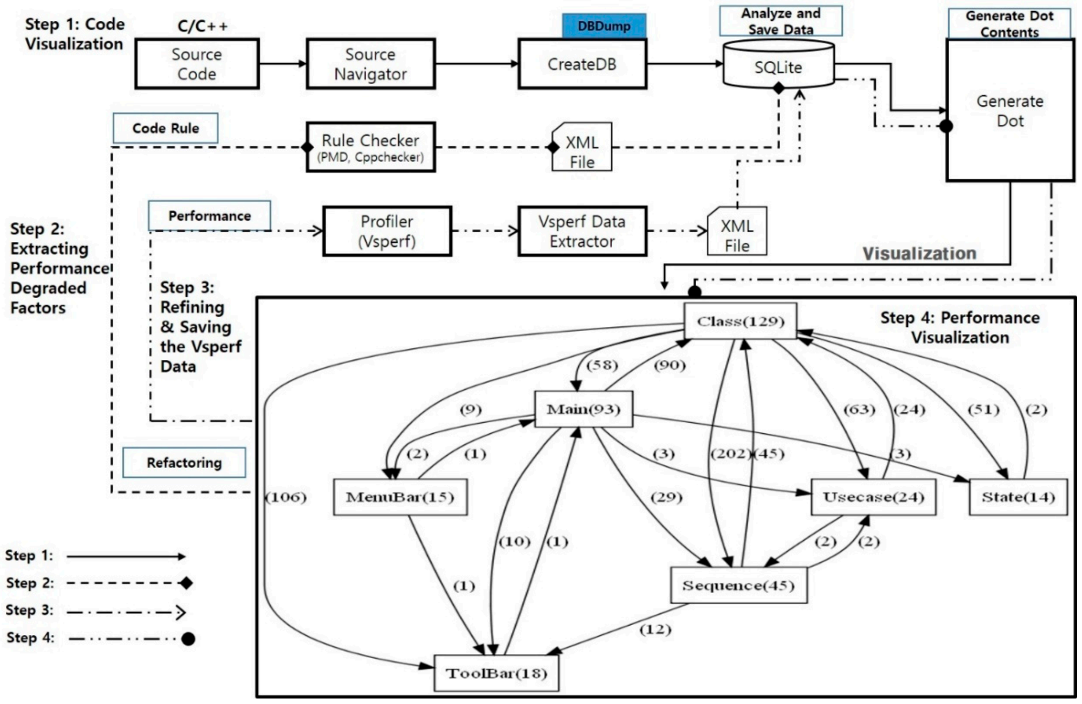

Figure 1. Performance improvement mechanism for procedural languages.

(1) Code Visualization

The phrase 'code visualization' means the visualization of internal code information extracted using a source navigator (SN). In the create database (CreateDB) command, we make storage space, SNDB.db, which stores the Source Navigator Database (SNDB) data in SQLite. This mechanism works as follows: (1) SN analyzes source code data; (2) SQLite stores them in database tables; and (3) a dot script generates an internal code structure graph. That is, the code visualization process is a serial order, such as input source code -> extract SNDB files -> extract C/C++ code data -> save DB Data -> apply code quality indicators -> code visualization.

We can extract the SNDB file with SN from the source code. SNDB includes the source code data that were analyzed by SN. The SNDB file is a binary file that should be converted to text using the database dump (DBdump) command. CreateDB extracts information from the code within DBdump. The data are stored in each database (DB) table. After this, we should create queries based on the code quality indicators for extracting the coupling, cohesion, and complexity and, especially, rule-checking against performance. Then, we measure our defined quantitative scores. Our visualization approach generates a code complexity graph from the data extracted with one of the queries, which identify high-complexity modules in the visualization graph. We also extract performance factors with RuleChecker and the VsPerf profiler [13].

(2) Extracting Performance Degraded Factors with Rule-Checking

We define the code performance's degraded patterns and rules with Cppcheck for visualization of CPS software performance [14,15]. In this paper, we just consider the following performance degraded element patterns: loop unrolling and loop down count, unnecessary control statements for inner loops, and multiple if-then-else statements. We also define these rules with the regular expressions through Cppcheck. Table 1 lists the code patterns and rules.

Table 1. Patterns and rules for performance degraded factors.

Pattern Name	Code Pattern	Regular Expression Rule

Loop unrolling and loop down count	int sum = 0;
	for (int i = 0; i < 1000; i++) {
	sum += array[i];
	}
		[a–z, A–Z, _] ([a–z, A–Z, _,0–9])
		* \&lt; ([0–9]) +;
		[a–z, A–Z, _] ([a–z, A–Z, _,0–9])
		* \+\+

Unnecessary control statements for inner loops	for(i = 0; i < 1000; i++){
	if(i & 0 × 01) {
	do_odd(i);
	}else{
	do_even(i);
	}
	}
		[a–z, A–Z, _] ([a–z, A–Z, _,0–9])
		* \&lt; ([0–9]) +;
		[a–z, A–Z, _] ([a–z, A–Z, _,0–9])
		* \+\+\)
		\{
		if\(

Multiple if-then-else statements	if (a == 1){
	}else if (a == 2){
	}else if (a == 3){
	}else if (a == 4){
	}
		if\(
		([a–z, A–Z, _,0–9, \b, \s, \+, V,
		\ *, \%%, -\&gt;, \&gt;, \&lt;,
		\[, \]]) * ==
		([0–9]) +\)

(3) Refining and Saving the VsPerf Profiler Data

Profiling of performance analysis is a dynamic form of program analysis. It analyzes parameters such as the time and space (memory) complexity of a program, the use of specific commands, and the frequency of function calls. Profiling data are further used to optimize the program code. Profiling can measure and analyze both source code and binary executable files.

VsPerf is a profiling tool provided by Microsoft Visual Studio, which runs directly from the Visual Studio integrated development environment (IDE), and measures the performance of an application. The performance analysis method of this tool has two elements: sampling and instrumentation [13].

Sampling interrupts the processor at desired intervals and collects the function call stack. It conducts a performance analysis without changing the executable files. However, if the call time for a particular function is less than the sampling interval, there may not be a single sample with that function. As a result, the sampling may contain a considerable data collection error. Instrumentation sets up a probe at the beginning and end of the function to be analyzed. It then obtains the function call stack and checks for events of all target functions. Although instrumentation can check the exact number of calls, it changes the executable file and introduces additional overhead owing to the probe. It must also analyze the relative time within the analysis information.

VsPerf provides a report on the code analysis results. The report types are process, function, call tree, and caller/callee. Reports include elapsed time, elapsed exclusive time, application-included time, application exclusive time, and call tree depth data.

(4) Performance Visualization

This is an important idea that we propose, to visualize the data stored in SQLite for the internal structure of the CPS C/C++ code, for easily identifying refactoring elements. The code performance information is included within the module box, with the module's name, the execution time, and the number of calls. This information compares the difference in performance between, before, and after refactoring. Figure 2 shows the methods view with each piece of execution information within a class name. The left box in Figure 2 displays method names and performance information (execution time and number of executions) of the particular class before refactoring. The right box displays performance information about a method of the same class after refactoring. The arrow indicates the current status of performance improvement as the time difference between, before, and after refactoring of the method. Through this process, we can clearly recognize whether it may have better performance or not. We show a performance improvement in our robot's simulator after refactoring at Section 4.

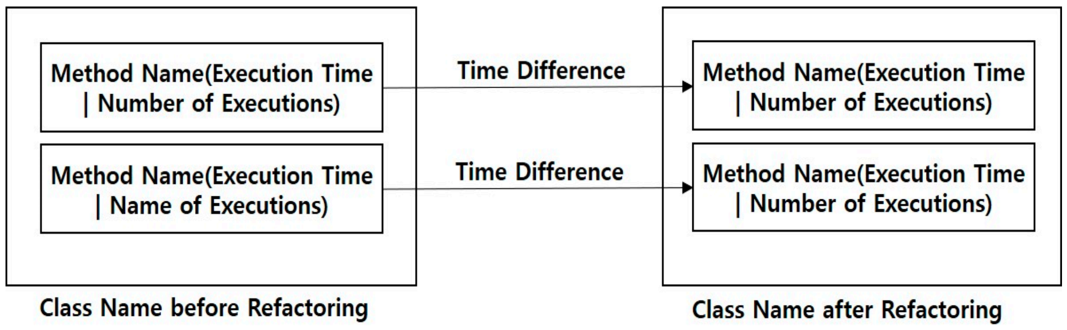
Method Name(Execution Time | Number of Executions)
Method Name(Execution Time | Name of Executions)
Time Difference
Time Difference
Method Name(Execution Time | Number of Executions)
Method Name(Execution Time | Number of Executions)
Class Name before Refactoring
Class Name after Refactoring

Figure 2. The class view.

4. An Exploratory Study

4.1. The Case of Original Source Code

In present day Korea, many researchers are focusing on autonomous robot simulators and vehicles for the fourth industrial revolution period. They are developing the hardware for their robots very well without software of high quality. Therefore, we consider how to enhance their software (simulators). Figure 3 shows a multiple-joint robot simulator. We apply this mechanism to a six-legged robot simulator. This simulator easily controls the motion of the multiple-joint robot with the controller within the tool. Even if a robot does not exist, its actions can be performed through tool simulations, i.e., the user creates an environment in which the simulated robot operates. The user enters the motion of the multiple-joint robot. The multiple-joint robot then operates in its environment [4,16]. We develop this multiple-joint robot simulator in C/C++.

Figure 4 illustrates the code architecture of the multiple-joint robot simulator. This architecture is the result of code visualization. In Figure 4, ① indicates that the module is unclear. The ② indicates a large class. The large classes are marked with the red color in the box. The ③ indicates strong coupling between modules and ④ indicates strong coupling with external modules. Finally, ⑤ shows the most strongly coupled modules among the visualization results. In the CSimulationView class of the red-colored box in ⑤ (72, 2418), 72 is the execution time, and 2418 is the number of executions. The class also has a red-colored incoming arrow, with [S * 7][CT * 2] = 34.8, where S means seven times the stamp coupling, CT means two times the content coupling, and the sum of them is 34.8.

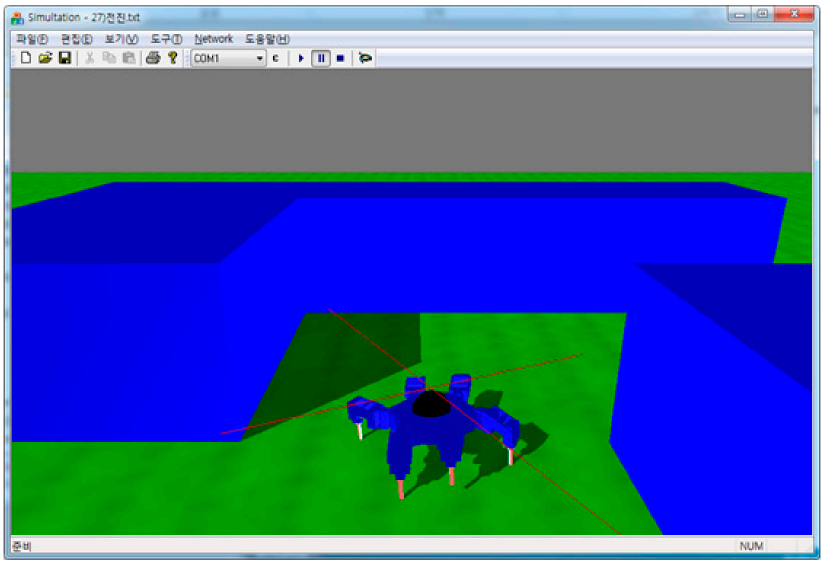
Simultation - 27)전진.txt
파일(F) 편집(E) 보기(V) 도구(T) Network 도움말(H)
COM1
준비
NUM

Figure 3. The multiple-joint robot simulator.

To optimize the modules, we set the extraction criteria based on the bad smell (a long method and a large class) for refactoring. For the extraction criteria, the number of methods was 20, and the sum of the number of method lines was 1000 or more. Here, the extracted module is displayed in the red color.

With this visualization of bad code patterns based on our defined quality indicators, we can easily guide the rewrite of the original code for better quality, which is refactoring.

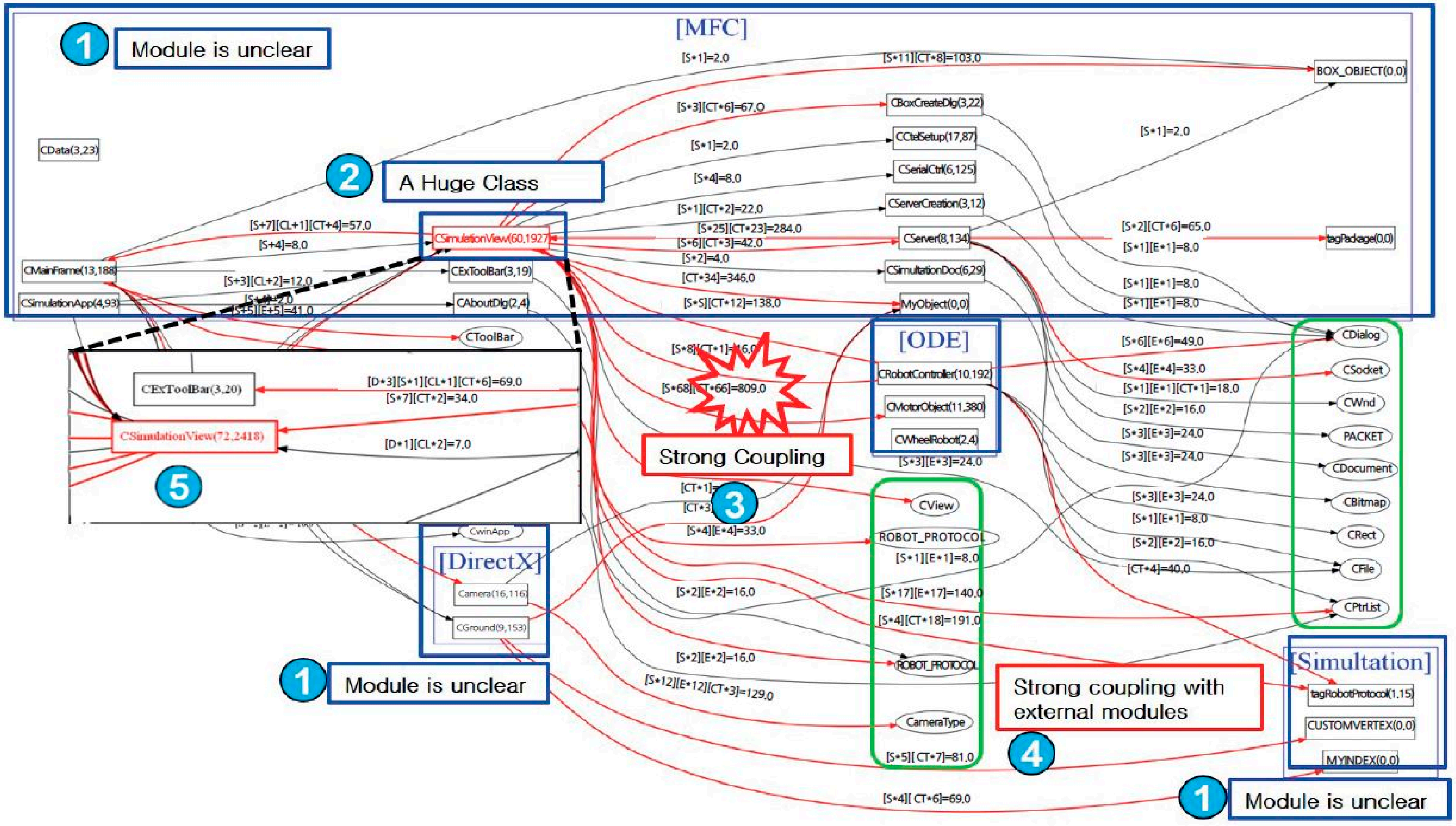

Figure 4. The inner code architecture of the multiple-joint robot simulator before refactoring.

4.2. The Case of Refactored Source Code

Our research focuses on reverse engineering, and we first statically analyze source code. We make queries with quality factors (such as coupling), and also store the analyzed results in DB tables. For example, coupling is composed of data, stamp, control, external, common, and content coupling. Here, the data coupling score is the lowest, and the content coupling score is the highest. In software engineering, the code's complexity is worse than the code size, that is, the internal structure of the software. The complexity of the code size is measured by its coupling, such as the calling and called modules. In this paper, we measure the coupling to assign adequate scores for each coupling. Table 2 shows the scores for each coupling.

Table 2. The scores for each coupling.

Coupling	**Data**	**Stamp**	**Control**	**External**	**Common**	**Content**
Score	1	2.4	3.9	5.6	7.5	10.2

The coupling formula is as follows:

$$\text{Total Coupling of Each Module} = \sum_{i=1}^{6}(Each\ Coupling\ Score_i\ \times The\ Number\ of\ Calls)$$

where, i = 1: Data, i = 2: Stamp, i = 3: Control, i = 4: External, i = 5: Common, and i = 6: Content.

Figure 5 presents the result of refactoring. The refactoring results are as follows. We modularize each part of the results, and improve the CSimulationView class that represents the highest level of complexity. We can also reduce the complexity. The two numbers in ① indicate the number of lines in the module and the number of module functions. The ② indicates the coupling value between modules. The code architecture represents the coupling between the modules as follows: D: Data, S: Stamp, CL: Control, E: External, CN: Common, and CT: Content. For example, the ② of Figure 5 shows the couplings between CSimulationView and the CServer module. Further, [CT * 21] = 10.2 * 21 = 214.0 on the directed red arrow where the number and the score of content (CT) couplings are 21 and 10.2. So, to reduce the code's complexity, we define the baseline of the sum of each coupling, that is, the score 30. On a static analysis of the source code, we automatically display the red colors of the arrow and module. When the score is greater than 30, we repeat refactoring again until the score is less than 30.

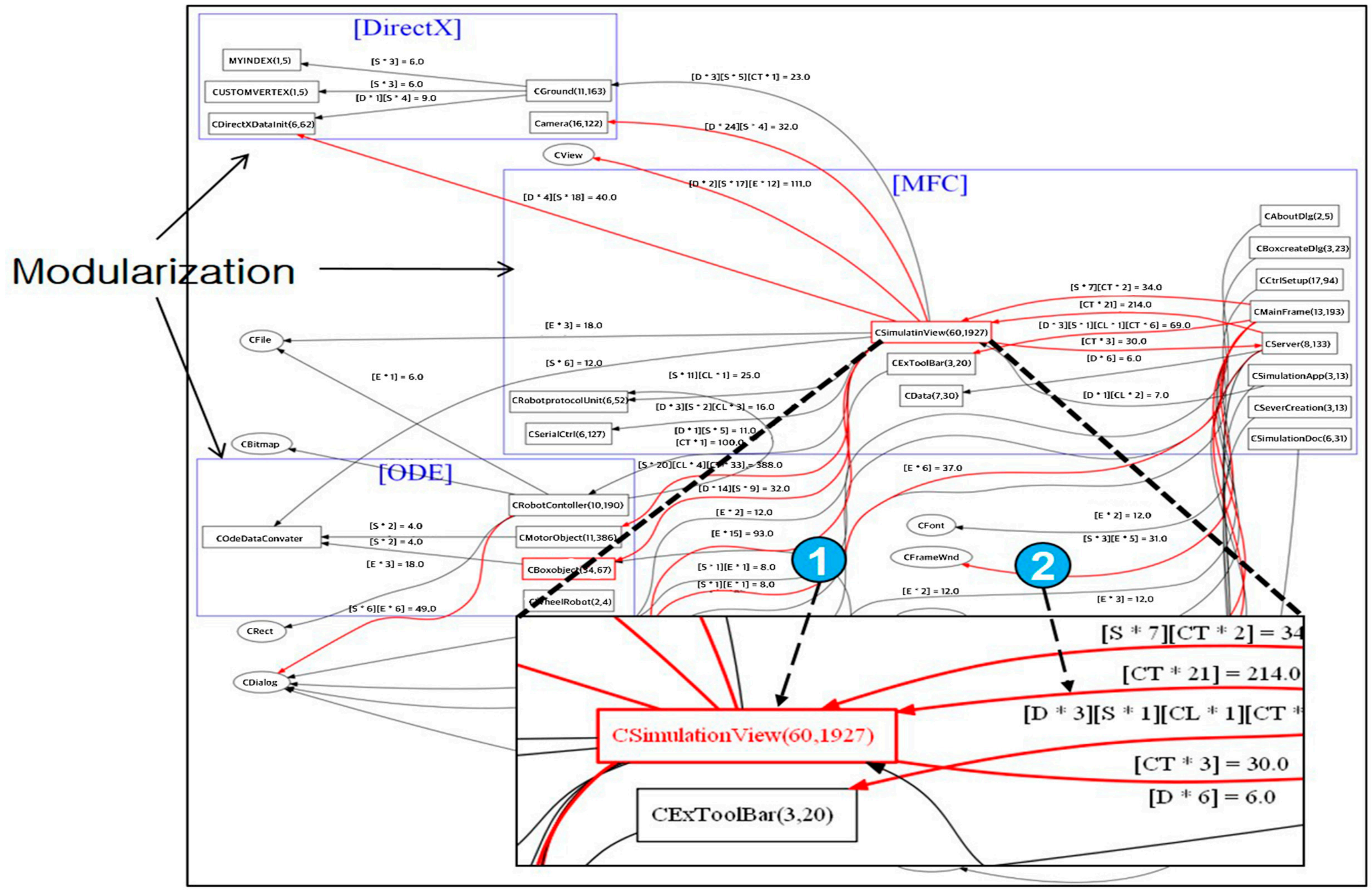

Figure 5. The inner code architecture of the multiple-joint robot simulator after refactoring.

Table 3 lists the results of the improvement in complexity of each module, including CExToolBar, CMainFrame, CGround, CSimulationView, CServer, CSimulation, and commerce eXtensible Markup Language (cXML), at each checkpoint time. However, it was not necessary to refactor the CMotorObject and CRobotController modules because of their low complexity values (0, 8). The second row of Table 3 shows the complexity values of the code modules before refactoring at the first baseline time. There are decreasing complexity values at each checkpoint time after refactoring. Figure 6 presents the results graph for the refactoring, which shows that low complexity values were achieved for all modules at the checkpoint three time. Figure 6 shows the decreasing slope of complexity values between checkpoint one and checkpoint two compared to the baseline.

Table 3. Refactoring results.

Coupling	Total	CExToolBar	CGround	CMain Frame	CMotor Object	CSimulation View	CServer	CSimulation	CRobot Controller	CXml
Baseline	5172.3	1133	49	950.4	0	1095.8	769	524.6	8	524.6
Checkpoint1	4226.9	187.6	49	950.4	0	1095.8	769	524.6	8	524.6
Checkpoint2	1258.1	187.6	49	91	0	114	97	69	8	69
Checkpoint3	1220.6	187.6	19	91	0	114	97	69	8	69

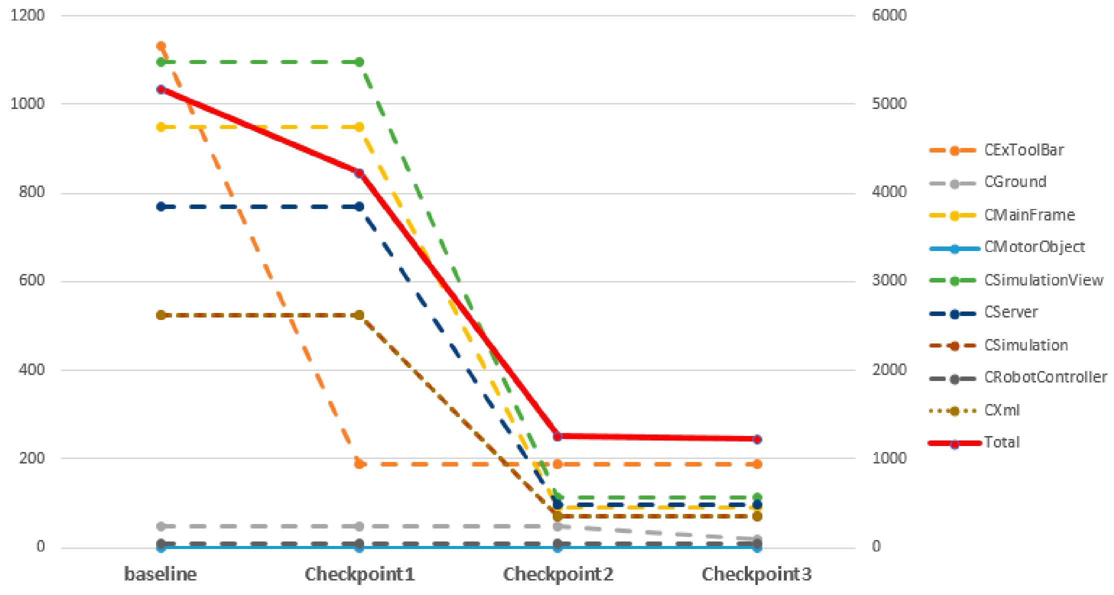

Figure 6. Refactoring results.

4.3. *Performance Results*

Figure 7 shows the results of the measurement of the multiple-joint robot simulator's performance. The class name (RobotSimulation) is displayed near the bottom of Figure 7. This result is a measure of performance by performing a refactoring of the CPS C/C++ code based on the baseline. Performance changes are divided into before and after clusters. In the most complex CSimulationView, each cluster represents the name, number of operations, and performance time of the methods inside the class. The arrows link the methods of each cluster. The figure above the arrow is the changed performance time. The performance measurement between before (①) and after (②) is "−45.21", i.e., the method after refactoring is 45.21 ms faster than before refactoring. The performance measurement between before (②) and after (③) is "−23,269.67", i.e., the method after refactoring is 23,269.67 ms faster than before refactoring. In the third refactoring, the performance measurement between before (③) and after (④) is "−68.66", i.e., the method after refactoring is 68.66 ms faster than before refactoring.

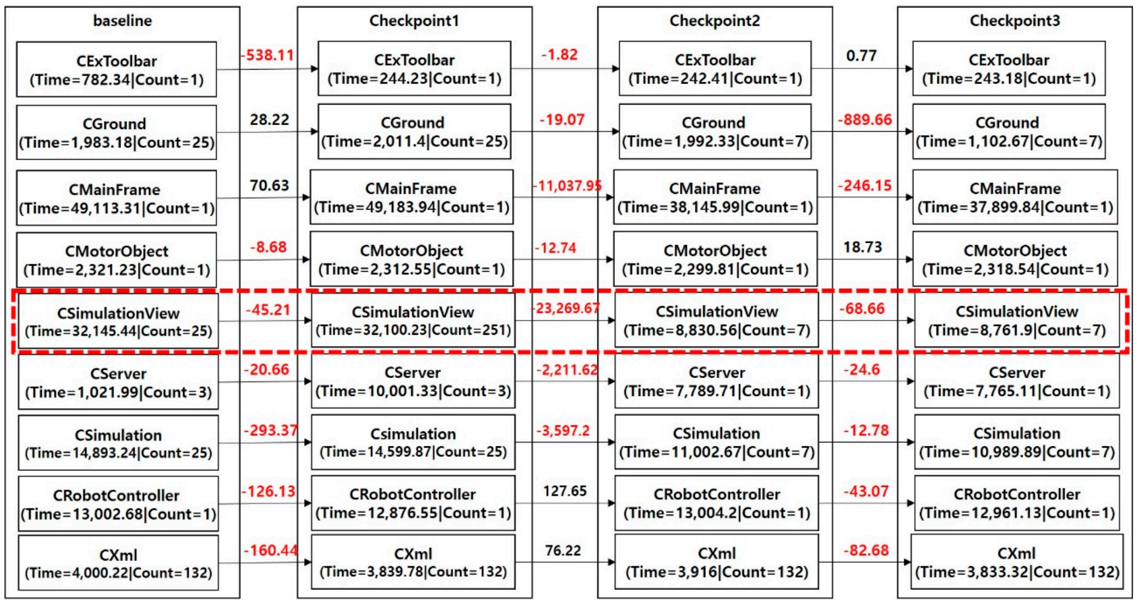
baseline
Checkpoint1
Checkpoint2
Checkpoint3
CExToolbar (Time=782.34|Count=1)
-538.11
CExToolbar (Time=244.23|Count=1)
-1.82
CExToolbar (Time=242.41|Count=1)
0.77
CExToolbar (Time=243.18|Count=1)
CGround (Time=1,983.18|Count=25)
28.22
CGround (Time=2,011.4|Count=25)
-19.07
CGround (Time=1,992.33|Count=7)
-889.66
CGround (Time=1,102.67|Count=7)
CMainFrame (Time=49,113.31|Count=1)
70.63
CMainFrame (Time=49,183.94|Count=1)
-11,037.95
CMainFrame (Time=38,145.99|Count=1)
-246.15
CMainFrame (Time=37,899.84|Count=1)
CMotorObject (Time=2,321.23|Count=1)
-8.68
CMotorObject (Time=2,312.55|Count=1)
-12.74
CMotorObject (Time=2,299.81|Count=1)
18.73
CMotorObject (Time=2,318.54|Count=1)
CSimulationView (Time=32,145.44|Count=25)
-45.21
CSimulationView (Time=32,100.23|Count=251)
-23,269.67
CSimulationView (Time=8,830.56|Count=7)
-68.66
CSimulationView (Time=8,761.9|Count=7)
CServer (Time=1,021.99|Count=3)
-20.66
CServer (Time=10,001.33|Count=3)
-2,211.62
CServer (Time=7,789.71|Count=1)
-24.6
CServer (Time=7,765.11|Count=1)
CSimulation (Time=14,893.24|Count=25)
-293.37
Csimulation (Time=14,599.87|Count=25)
-3,597.2
CSimulation (Time=11,002.67|Count=7)
-12.78
CSimulation (Time=10,989.89|Count=7)
CRobotController (Time=13,002.68|Count=1)
-126.13
CRobotController (Time=12,876.55|Count=1)
127.65
CRobotController (Time=13,004.2|Count=1)
-43.07
CRobotController (Time=12,961.13|Count=1)
CXml (Time=4,000.22|Count=132)
-160.44
CXml (Time=3,839.78|Count=132)
76.22
CXml (Time=3,916|Count=132)
-82.68
CXml (Time=3,833.32|Count=132)

Figure 7. Results of the measurement of the multiple-joint robot simulator's performance.

Table 4 lists the performance results, whereas Figure 8 shows the analyzed results of performance on refactoring. In Table 4, the baseline row indicates the performance value of the code before refactoring, whereas each checkpoint row indicates the performance value after refactoring. At each checkpoint, we show the improvement in performance with the difference between the performances of each module compared to the baseline.

Table 4. Performance results.

Coupling	CExToolBar	CGround	CMainFrame	CMotor Object	CSimulation View	CServer	CSimulation	CRobot Controller	CXml
Baseline(ms)	782.34	1983.18	49,113.31	2321.23	32,145.44	10,021.99	14,893.24	13,002.68	4000.22
Checkpoint1(ms)	244.23	2011.4	49,183.94	2312.55	32,100.23	10,001.33	14,599.87	12,876.55	3839.78
Checkpoint2(ms)	242.41	1992.33	38,145.99	2299.81	8830.56	7789.71	11,002.67	13,004.2	3916
Checkpoint3(ms)	243.18	1102.67	37,899.84	2318.54	8761.9	7765.11	10,989.89	12,961.13	3833.32

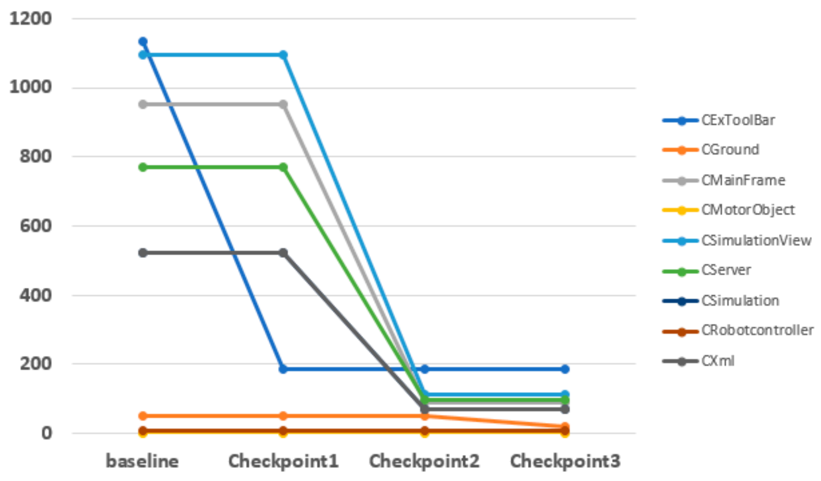

Figure 8. Results of performance measurement.

The complex and extensive CPS C/C ++ code can confirm the change in the software's performance through the proposed method. Furthermore, our code visualization can improve the performance of the CPS-based C/C ++ code. Consequently, we can improve the quality of the CPS-based procedural language, that is, C/C++ code.

5. Conclusions

A CPS may be a networking system with optimal performance in a system wherein computers and the real world interact. A CPS monitors the state of a physical system with various embedded devices. It further performs a desired operation on the data collected by the monitoring, which affects the computation result of the CPS for the system. Simulation techniques are used to verify CPS software. However, a CPS works simultaneously for various embedded systems, which considerably increases the code size and complexity of CPS-based software. Furthermore, CPS software does not guarantee good performance or quality. We visualized the performance measurement of CPS-based C/C ++ code to rewrite the C/C ++ code, using the proposed method, and performed a dynamic analysis with VsPerf, for efficient CPS software improvement. The analysis involved the actual CPU and memory usage. We extracted the complexity of the CPS software through code visualization, identified a high-complexity module (a core algorithm), and then finally measured the performance to improve it. We showed developers how to identify memory performance indicators and inner structures for CPS software. In the future, we will include the power consumption pattern of software, find patterns of power-consumption in the source code, and then visualize these patterns.

Author Contributions: B.K.P., B.J., and R.Y.C.K. designed the present study, checked the literature, and wrote the manuscript; B.K.P. and R.Y.C.K. performed the static analysis and code visualization; B.K.P. and R.Y.C.K. critically revised the manuscript; all authors gave the approval for the final version of the review submitted for publication. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Basic Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2017R1D1A3B03035421 and NRF-2017R1D1A3B03034102), and also supported by the Ministry of Trade, Industry and Energy(MOTIE), KOREA, through the Education Program for Creative and Industrial Convergence (Grant Number N0000717).

Conflicts of Interest: The authors have no conflict of interest.

References

1. Lee, J.; Bagheri, B.; Kao, H.-A. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manuf. Lett.* **2015**, *3*, 18–23. [CrossRef]
2. Jin, H.; Park, T. *The Base System of the Fourth Industrial Revolution*; Software Policy & Research Institute Magazine; CPS, Monthly Software Oriented Society: London, UK, December 2016.
3. Kim, W.; Jeon, I.; Lee, S.; Park, S. CPS Technology Trends. *Wkly. Technol. Trends* **2010**, *1*, 1–10.
4. Son, H.S.; KIM, R.Y.C. The Pre-Testing for Virtual Robot Development Environment. *IEICE Trans. Inf. Syst.* **2018**, *101*, 1541–1551. [CrossRef]
5. Son, H.S.; Kim, W.; Jeon, I.; Lee, H.; Jeon, J.; Kim, R.Y. A Method of Representing Topographic Environment Data through SEDRIS in Cyber Physical System. *Korea Soc. Simul. Mag.* **2011**, *1*, 11–18.
6. Koziolek, H. *Performance Evaluation of Component-based Software Systems: A Survey*; Performance Evaluation; Elsevier: Amsterdam, The Netherlands, 2010.
7. Liu, H.H. *Software Performance and Scalability, A Quantitative Approach*; Wiley: Hoboken, NJ, USA, 2009.
8. Kang, G.-H.; Kim, R.Y.C.; Lee, J.H. A Case Study on Performance Improvement through extracting Software Performance Degradation Factors. *Int. J. Appl. Eng. Res.* **2015**, *10*, 90.
9. Park, B.K.; Kang, G.; Kim, R.Y.C. Performance Measurement of Procedural Code for CPS Multiple-Joint Robotics Simulator. *Adv. Eng. ICT-Converg. Proc. (AEICP)* **2019**, *38*, 75–78.
10. Park, B.K.; Kang, G.; Kim, R.Y.C. Software Visualization Approach for Performance Measurement of Object-Oriented Code based on Cyber-Physical Systems (CPS) Software. *Adv. Eng. ICT-Converg. Proc. (AEICP)* **2019**, *38*, 28–30.
11. Park, J.; Son, H.S.; Kim, R.Y. Developing an Automatic Tool for Visualizing Source Code against Bad Smell Patterns. In Proceedings of the Global Conference on Engineering and Applied Science, Okinawa, Japan, 7–9 July 2017.
12. Moon, S.Y.; Park, B.K.; Kim, R.Y.C. Code Complexity on Before and After Applying Design Pattern through SW Visualization. In Proceedings of the 2016 International Conference on Platform Technology and Service (PlatCon), Jeju, Korea, 15–17 February 2016.

13. VsPerf. Available online: https://docs.microsoft.com/en-us/visualstudio/profiling/vsperf?view=vs-2017 (accessed on 24 December 2018).
14. Kang, G.; Kim, R.Y.C.; Lee, S.E.; Jeon, S.N. Extracting performance factors against performance degradation through Code Visualization. In Proceedings of the International Conference on Convergence Technology, Hokkaido, Japan, 29 June–2 July 2015; pp. 276–277.
15. CPPCheck. Available online: http://cppcheck.sourceforge.net/ (accessed on 10 January 2019).
16. Son, H.S.; Kim, W.Y.; Kim, R.Y. Semi-Automatic Software Development Based on MDD for Heterogeneous Multi-Joint Robots. In Proceedings of the International Symposium on Control and Automation, Sanya, China, 13–15 December 2008.