Identifying and Measuring Modularity Violations in Cyber-Physical Systems

Lu Xiao and Michael J. Pennock
School of Systems and Enterprises
Stevens Institute of Technology
{lxiao6, mpennock}@stevens.edu

***Abstract*—In recent years, complex cyber-physical systems (CPS) have achieved widespread application in diverse areas. The hardware and software components in CPS are deeply intertwined at various levels of abstraction under changing contexts to achieve the desired goals. One way to manage this complexity is through the use of modular architecture that enables portions of a CPS to be upgraded, replaced or fixed in a plug-and-play manner. However, is it really modular such that components can be independently replaced? This suggests a need for methods to identify modularity violations in cyber-physical systems. In this paper, we conducted a case study of a large-scale open source CPS: OpenWrt, which is a common software infrastructure for the Internet of Things. What we found was that while the software architecture of OpenWrt is well-modularized, there were a number software modularity violations that appear to involve hardware-related concepts. Furthermore, we found that software components related to hardware were much more likely to change frequently. Combined, these findings suggest hardware components are a source of latent modularity violations in OpenWrt and demonstrate the feasibility of adapting methods to identify and measure modularity violations from pure software systems to cyber-physical systems.**

***Index Terms*—Modularity violation, Cyber-physical system, change propagation, hardware, software**

I. Introduction

In recent years, cyber-physical systems (CPS) have achieved widespread application in diverse areas including: civil infrastructure, energy, health-care, transportation, automotive, smart appliances, and others [14]. Usually cyber-physical systems are composed of diverse subsystems consisting of both physical and software components developed by different vendors. These components are deeply intertwined at various levels of abstraction (e.g. data flow, physical connections, and logical connections etc.) under changing contexts to achieve the desired functionalities and the respective quality attributes, such as performance, security, scalability, maintainability, etc. With the advance of technology, the recognition of new consumer needs, and the detection of deficiencies in current systems, components need to be upgraded, replaced, and fixed—frequently, in many domains. The key question is: can the upgrade, replacement, or problem fix happen quickly and without disrupting the rest of the system?

To address this issue, stakeholders have increasingly emphasized modular and open approaches to system development. According to Baldwin and Clark [2], modularity allows for both independence of structure and integration of functions in large and complex systems. The goal of modularity is to improve interoperability, facilitate system evolution and technology insertion, and foster competition. However, it can be difficult for the stakeholders to assess whether the resulting architectures and systems are truly modular. In other words, can modules in a CPS actually be upgraded, replaced, and fixed in a plug-and-play manner without affecting one another to maximally leverage the benefits of modularization? In the traditional software engineering field, it has been observed that this is often not the case. Modules without any explicit structural dependencies were seen frequently changing together when new features or bug fixes were implemented. Wong et. al. [15] call this phenomenon a Modularity Violation (MV). Studies of real-life open source projects indicate that up to 85% of bug-fixing efforts in a software system involve modularity violations [20]. An in-depth analysis revealed that modularity violations usually result from and imply latent connections among software modules. For example, it could be an implicit assumption regarding the usage of time units in two modules [16].

In software systems, the consequences of modularity violations could be higher bug-rates (when the assumptions are not consistent among different parties) and increased maintenance costs (in order to keep the assumptions consistent) [1], [20]. It is possible that that modularity violations could be even more harmful in cyber-physical systems. Modularity violations, derived from the latent relationships among software and hardware components, prevent the long-term evolution, maintenance, and success of cyber-physical systems. In particular, due to the inflexibility and high cost of evolving hardware modules, modularity violations in cyber-physical systems could more easily lead to vendor-lock-in compared to a pure software environment.

However, there still is a lack of techniques and empirical experience that would allow stakeholders to detect, measure, and understand modularity violations in developed and acquired cyber-physical systems. Since the term first emerged in 2006, it has been recognized that the design, modeling, and maintenance of cyber-physical systems is more challenging than for traditional engineering systems, due to the intrinsic heterogeneity and nondeterministic nature of the interactions among cyber and physical components [4], [9], [12]. Existing engineering techniques either focus on the cyber side or the physical side, but not both. This paper tackles one aspect of

this problem by identifying the latent connections between the cyber and physical aspects of the system. More specifically, it aims to identify the shared concepts that potentially propagate maintenance actions between the cyber and the physical sides. Our results have shown the feasibility of this approach, and revealed that it is more likely that the direction of propagation is from the physical side to cyber side. Therefore, this paper is the first to offer stakeholders a new keyword-based approach and empirical experience to understand how the cyber and physical sides interact and impact each other.

In this paper, we organize modularity violations in cyber-physical systems into three different types: 1) software vs. software, 2) software vs. hardware, and 3) hardware vs. hardware. The first type of modularity violation has been extensively investigated in prior work [1], [15]–[17], [20]. There, the approach to identify and measure modularity violations relies on the analysis of source code and operational data such as maintenance activity records. These are automatically and comprehensively tracked in version control systems. The challenge to extending this type of analysis to identify modularity violations involving hardware is the lack of systematic records of operations involving hardware components. Unlike, software projects that use standard version control systems to keep track of who made what changes to which parts at what time and for what reason, changes to hardware components are not always tracked in similar detail or are not as easily accessible. Given these limitations, the question naturally follows, could a subset of hardware modularity violations be inferred from an analysis of records of software changes? More specifically, some software changes may be a consequence of hardware related modularity violations. If that were the case, analysis of version control systems could be used to detect some hardware related modularity violations outright and flag other potential violations for further investigation. The issue is whether or not there is actually enough information in a version control system to infer a hardware modularity violation.

In this paper, we conducted a case study of an open source cyber-physical system: OpenWrt, which aims to develop a common software infrastructure for the Internet of Things (IoT). To analyze these systems, we applied a keyword-based heuristic to help us identify software-hardware modularity violations, where the software artifacts change due to hardware related issues. We identified 70 software source files that are potentially involved in software-hardware level modularity violations because the changes made to these files involved hardware related concepts. We conjecture that hardware-related concepts that propagate changes to software artifacts imply a potential modularity violation at the software-hardware level. Consequently, the case study demonstrated the feasibility of identifying potential modularity violations in a cyber-physical system by analyzing a software version control system. We made three observations through this case study:

- The OpenWrt is more modularized than about 85% of the 129 (commercial and open source) traditional software systems we studied before.
- Software-Software modularity violations in OpenWrt include hardware related information in the naming conventions of the involved source files.
- Hardware-related concepts are the main contributing factors to software-hardware modularity violations in OpenWrt.

II. Background

A. Modeling Cyber-physical Systems

The term cyber-physical systems emerged around 2006, when it was coined by Helen Gill at the National Science Foundation [10], [12]. Gill defined a cyber-physical system (CPS) as an integration of computation with physical processes. However, one of the challenges to designing and managing cyber-physical systems is that there are techniques to represent either the cyber processes or the physical processes, but not both [8]. From the cyber perspective, there are different techniques to represent the architecture of software systems, such as architecture description languages, UML models, and component models. From the physical perspective, there are different traditional engineering techniques to model the development of physical systems. One study on architecting cyber-physical systems found that, currently, researchers tend of focus on a specific attribute of interest rather than assessing the cyber-physical architecture as a whole [19].

However, there are some instances of research that take a more holistic approach to architecting cyber-physical systems. Rajhans, et al. [6] contributed a new cyber-physical architectural style to present the interconnections and interactions between physical and cyber components. They defined three related families of general components and connectors pertaining to the cyber domain, the physical domain, and their interconnections. From the architectural assessment perspective, Sinha [7] developed a theoretical framework for structural complexity quantification and its implications for the design of cyber-physical systems. Derler et al. [10] focused on the challenges of modeling cyber-physical systems that arise from the intrinsic heterogeneity, concurrency, and sensitivity to timing of such systems. They described some promising approaches that use domain-specific ontologies to enhance modularity and jointly model of the functional and implementation architectures. Finally, Cristalli et. al. [11] provides a representative example of a modular approach to developing a cyber-physical system through their modular design for a Smart Robotic Cell, composed by several interconnected subsystems.

What all of the above have in common is a focus on designing cyber-physical systems to be modular. However, there is no guarantee that the realized system will exhibit the intended modularity. The need to assess the actual modularity of a cyber-physical system is particularly acute. As Lee [13] points out, cyber-physical systems have always been held to a higher reliability and predictability standard than general-purpose computing. Without reliability and predictability, cyber-physical systems will not be applied in safety-critical domains like traffic control, automotive safety, and healthcare. While a modular design is just one approach to im-

prove reliability, predictability, and maintainability; when it is employed, one would like to know how well that was achieved. While approaches to assess modularity in pure software systems have been developed, to the best of the investigators knowledge no approaches have been developed that specifically address the unique challenges of assessing the modularity of cyber-physical systems. Thus, the question is whether existing approaches from the software domain can be adapted to the cyber-physical domain.

B. Detecting Modularity Violations in Software Systems

The term modularity violation was first proposed by Wong et al. [15] to describe the phenomenon where independent software modules frequently change together during the evolution of a system in the process of fixing bugs or adding features. Methodologies and tools have been built for analyzing modularity violations in software systems [1], [15], [17]. The identification of a modularity violation follows three steps:

1) Reverse-engineering the modular structure of a software system from the code base: The modular structure of a software system can be calculated based on the structural dependencies among software entities. Xiao et al. developed a new architectural representation, called Design Rule Space (DRSpace) modeling to capture the modular structure of a software system as multiple overlapping design spaces [17], applying Baldwin and Clark's [2] design rule theory. The modular structure can be visualized in the form of a DSM (Design Structure Matrix). The items in the DSM represent software entities, the cells represent the structural dependencies among entities. The entities can be clustered into modules based on selected dependency types for supporting analysis of different focuses.
2) Extracting the evolutionary coupling (i.e. the number of co-changes) between software modules: This can be calculated based on the data recorded in version control systems, which automatically keep track of all the changes made to software entities, in terms of who at what time made what changes to which entities/modules for what reason. Analyzing such data helps to extract the evolutionary coupling between any two software entities: how many times they are modified together in the course of maintenance activities. The more frequently two entities are modified together, the stronger is their evolutionary coupling—implying latent connections.
3) Calculating the discrepancy between the above two data sets to point to modularity violations among software modules: If two software entities/modules are structurally independent according to step 1, but they have high evolutionary coupling according to step 2, there is a potential modularity violation. According to prior empirical studies of industry software systems, modularity violations usually indicate implicit assumptions shared among software modules. For example, it could be an implicit assumption regarding the usage of time units in two modules [16].

In software systems, the consequences of modularity violations could be higher bug-rates (when the assumptions are not consistent among different parties) and increased maintenance costs (in order to keep the assumptions consistent) [1], [17], [20]. However, the above-mentioned approach is limited in that it only addresses software systems implemented in a single programming language. Complex hybrid systems are composed of multiple heterogeneous hardware and software sub-systems. Such systems are particularly difficult to analyze because of data inconsistency and heterogeneity among the various sub-systems. The goal of this study was to test the feasibility of leveraging software maintenance data to infer hardware related issues that are indicative of software-hardware modularity violations.

III. Approach

As discussed earlier, the identification of modularity violations in a CPS is a challenge because maintenance operations on hardware components are not as comprehensively recorded as in traditional software systems. However, we hypothesized that software components that are changed frequently due to hardware related concepts are indicative of potential modularity violations. As a case study, we chose a real life software infrastructure, OpenWrt, for cyber-physical systems. OpenWrt is a Linux based core that is commonly used to support the Internet of Things (IoT). Table I summarizes basic facts of these projects. It shows the scale (measured by number of files, methods, and lines of code), the development team size, and the age (measured by the number of revisions and starting time). We chose this project because it is an industry scale project offering sufficient research data.

TABLE I: Case Study Subject

Project Name:	OpenWrt
# of Files:	1052
# of Methods:	6061
# of Lines of Code:	163114
# of Developers:	137
# of Revision Records:	38099
Range of History:	2004 to 2017

Our case study followed three steps:

1) We recovered the modular structure of OpenWrt using reverse engineering techniques. Specifically, we leveraged the tool set previously build by Xiao et. al. [17]. The input of this step included the code repository and code revision history. This step generated two Design Structure Matrices: one for the static structural dependencies and the other for the dynamic evolutionary coupling as shown in Figure 1. The detailed interpretation of these two matrices will be elaborated later.
2) We measured software level modular structure using a metric called *Decoupling Level*. This metric describes how well a system is decoupled into small and manageable modules [1]. In particular, we compared the

decoupling level of OpenWrit with 129 traditional software systems, which Mo. et. al. calculated before. This facilitated the understanding of the overall modular structure of OpenWrt, as well as how modularized it is relative to other systems.

3) We developed and used keyword-based heuristics to identify and measure hardware related modularity violations in OpenWrt. We manually extracted hardware related keywords appeared in the revision message (usually a brief narrative explaining the rationale/goal of a change). Then, we identified source code regions that were revised due to hardware-related concepts. The identified source code regions are likely to be involved in modularity violations with hardware components.

IV. Results

In this section, we will discuss the results of each step and the three main observations we made on OpenWrt.

1) The modular structure reversed from OpenWrt's code base indicates that it is more modularized than about 85% of the 129 (commercial and open source) traditional software systems we studied before.
2) There are 23 source files in OpenWrt involved in software level modularity violations. Most of these files include hardware related information in the naming conventions, implying hardware concepts are the underlying causes of these software module co-changes.
3) Using manually extracted key words, we identified 70 source files involved in software-hardware level modularity violations. Hardware-related concepts are the main contributing factors to such modularity violations in OpenWrt.

A. Measuring OpenWrt Modular Structure

First, we wanted to understand how well OpenWrt is modularized. To do that, we reverse-engineered the code base of the projects. Figure 1a shows an overview of the modular structure identified in OpenWrt, represented in the form of a DSM [2]. This DSM is a square matrix showing the structural dependencies among source files in OpenWrt. The rows and columns represent source files, and a black dot represent a structural dependency from the row to the column. Figure 1a is just a qualitative overview of the interdependencies among the source files in OpenWrt, without capturing the details, such as file names and dependency types.

To get quantitative understanding of how modularized OpenWrt is based on the structural dependencies among source files, we calculated two metrics: 1) the decoupling level (DL) and 2) the propagation cost (PC), based on the DSM.

Decoupling level proposed by Ran et. al. [1] measures how well a system is decomposed into small and manageable modules that can evolve independently from each other. The highest possible value is 1, meaning the system is perfectly modularized (which is not likely in practice). The lower the DL value is, the less modularized a system is. For example, if there are 5 modules in a system and these 5 modules are completely independent from each other, the DL metric of such a system will be 1. In comparison, if the 5 modules are completely interconnected, the DL value will be 0. The DL value for OpenWrt is 0.78. Figure 2 shows the distribution of the DL value for 129 projects (108 open source projects and 21 industrial projects) from our prior study [1]. The red star in Figure 2 marks the position of the DL of OpenWrt compared to these 129 projects. The data shows that the DL value of OpenWrt is higher than about 85% of the 129 software projects. The implication is that OpenWrt is better decoupled into mutually independent modules compared to the majority of the previously studied software projects.

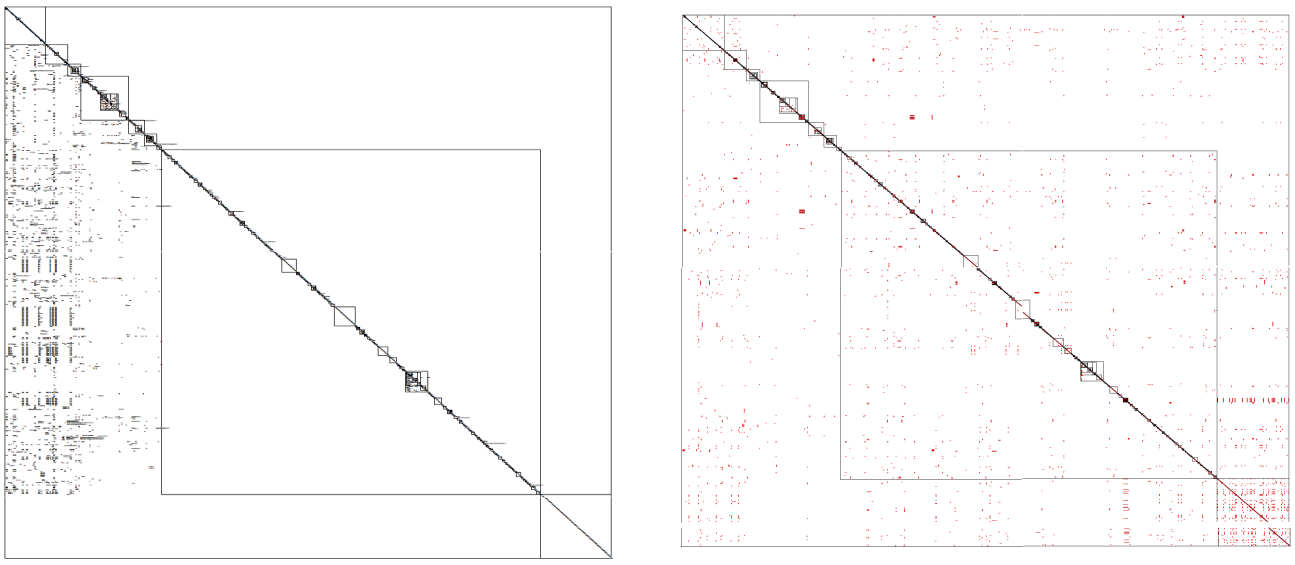
Similarly, we also calculated the propagation cost metric proposed by MacCormack et. al. [5]. Propagation cost measures the density of the n-transitive closure of the DSM. The maximum value is 100%, meaning that every element is connected (directly or transitively) to another element in the system. The lower the value, the more independent the modules are. Decoupling level and propagation cost are negatively correlated. That is the higher the DL, the lower the PC, and vice versa. The result shows that the PC value for OpenWrt is only 1.4%, which is lower than about 90% of the 129 (commercial and open source) traditional software systems we studied before. The Propagation Cost, again, suggests that OpenWrt is well modularized compared to other software systems we studied in the past.

In summary, the first observation is that the software components in this particular CPS are more modularized compared to traditional software systems. We conjecture that this is because the hardware components in a CPS increase the overall complexity of the system relative to pure software systems. Increasing the modularity of the software may be a way to cope with that complexity. Furthermore, by talking to IoT experts, we realized that the first priority of the software components is to stay concise to reach the quality goals of energy and cost efficiency in IoT systems.

B. Identifying Software-Software Level Modularity Violations

We applied the approach described in Section II to identify software-software level modularity violations [1], [15], [17]. The goal is to identify the potential contributing factors to software level modularity violations in CPS's. More specifically, we aim to determine whether software-software level modularity violations in CPS systems imply hardware related issues. If so, the hardware concepts are potential causal factors of modularity violations among software components in a CPS.

First, we calculated the evolutionary coupling among source files in OpenWrt in order to identify the modularity violations. The evolutionary coupling between two source files is the number of times the two files change together in the revision history. The overview of the evolutionary coupling among files in OpenWrt is illustrated in Figure 1b. Similar to Figure 1a, the rows and columns represent the source files in OpenWrt. Each red dot indicates the evolutionary coupling between the file on the row and the file on the column. Actually, we



(a) Modular structure

(b) Evolutionary coupling

Fig. 1: OpenWrt modular structure and evolutionary coupling

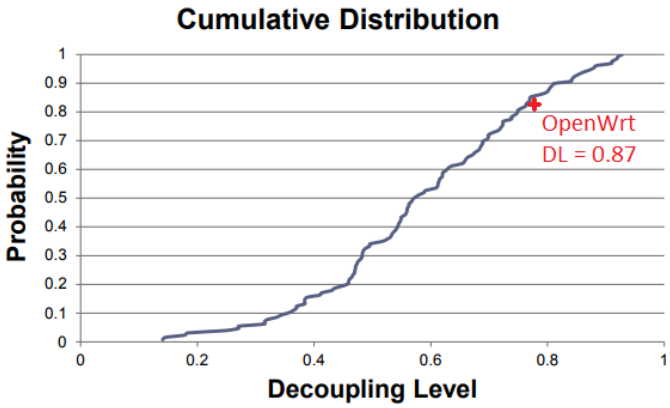


Fig. 2: OpenWrt decoupling level compared to other systems

observe that the weight of most of the evolutionary coupling is below 4, indicating software source files do not change together frequently. In particular, this is tracked from more than a decade of revision history (between 2004 and 2017) of OpenWrt. The evolutionary coupling is much lower compared to other traditional software systems we studied in the past.

Second, we computed the discrepancy between the structural dependencies shown in Figure 1a and the evolutionary coupling in Figure 1b. To focus on strong evolutionary coupling, we set the evolutionary coupling threshold to be 4. This is because it is common for two independent modules to change together for accidental reasons. The result is displayed in Figure 3, containing 23 source files that exhibit evolutionary coupling above 4, but are structurally independent from each other. The number in each cell shows the number of times two files change together in history (i.e. evolutionary coupling). For example, *cell[10, 6]* says "5", indicating file 10, *mach-nbg6716.c*, and file 6, *mach-wlr8100.c* change together 5 times in the revision history. We consider these 23 source files as potential modularity violations, which indicates shared but latent assumptions among them.

Further manual inspection of the source files involved in modularity violations indicated that hardware-related concepts contributed to these software level modularity violations. For example, the naming conventions of the source files imply that these files share hardware-related concepts. As highlighted in Figure 3, from row 1 to row 10, the naming of the files all contain mips, which is a typical microprocessor architecture for CPS. In addition, row 20 to 23 shows that the file names all contain firmware, which is held in non-volatile memory devices, such as ROM or flash memory. Based on these observations, we conjecture that the hardware concepts are actually the causal factors of these software-software level modularity violations in OpenWRT.

C. Identifying Hardware-Software Level Modularity Violations

Reasoning based on the above observation, we infer that hardware related concepts could also be the contributing factors for software-hardware modularity violations. That is, software components that change frequently due to hardware related concepts. As an example, we find this comment when developers changed a software entity: *"set chip type directly in ar8216_id_chip"*, where chip is obviously a hardware term. We assume it is less likely the other way around: software concepts contribute to hardware changes, because it is, in most cases, more affordable to change software components.

We manually extracted 16 key words from the commit messages and hardware devices supported by OpenWrt. The details are shown below in Table II, containing keywords, like radio, WiFi, zigbee, etc. By matching these manually extracted

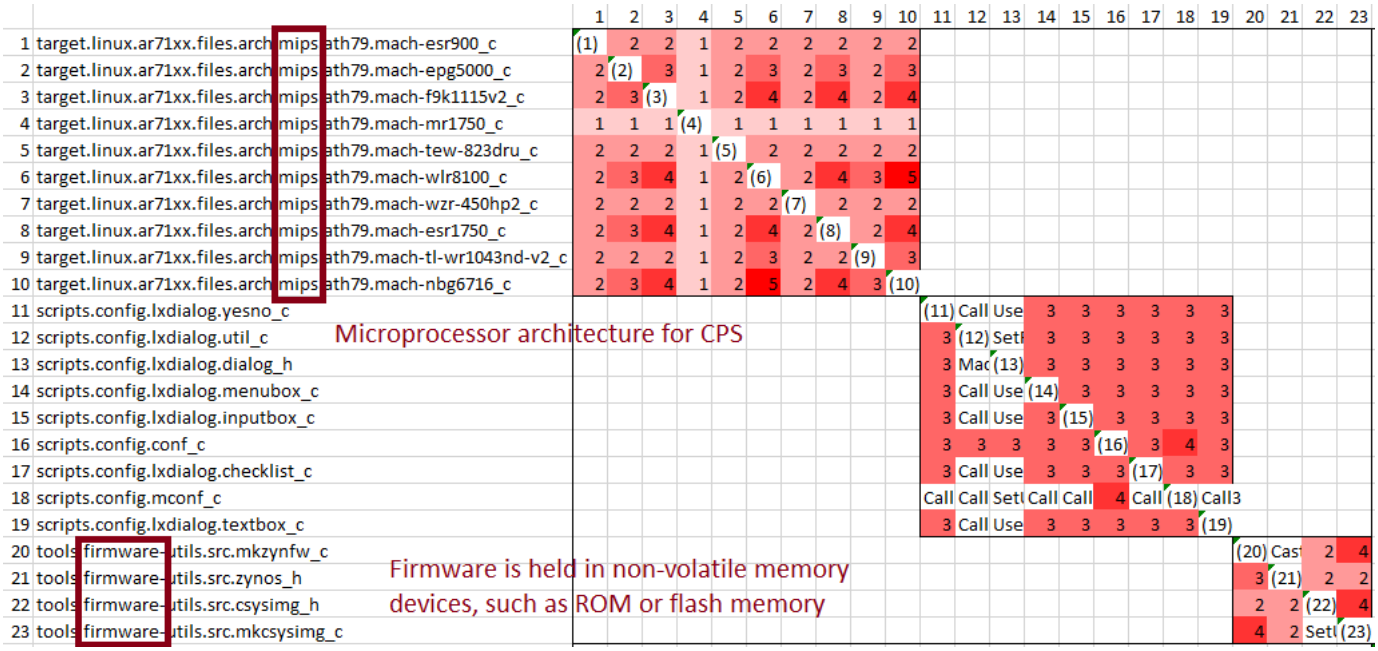


Fig. 3: Software-software modularity violations

keywords in developers change comments (which are usually used to explain why they made the change), we identified 70 source files in OpenWrt (of the 1052 total files) are potentially involved in software-hardware modularity violations.

TABLE II: Hardware keywords in OpenWrt

”radio”, ”WiFi”, ”zigbee”, ”btle”, ”mips”, ”ramips”, ”mtd”, ”broadcom”, ”routerboot”,”router”, ”firmware”, ”bluetooth”, ”energy”, ”power”, ”soc”,

To further understand the significance of hardware-related concept as change contributors, we calculated the change-proneness ranks of the 70 files involved in hardware-software level modularity violations, and we compare this with the change-proneness ranks of other general source files. The comparison result is shown in Figure 4. The x-axis represents the 10 ranks of change-proneness levels in a percentile scale. For example, 10% means source files rank in the top 10 most change-prone percentile; while 100% basically means all the files that have ever been changed in history. The y-axis represents the cumulative probability of a file residing above the respective change-prone rank percentile on the x-axis. In Figure 4, the upper and lower lines are associated with hardware-software modularity violations files and with all the changed files in the project respectively. For instance, the data shows files in hardware-software modularity violations have 16% chance to rank in the top 10% most change prone percentile; while a general file (among the total 657 files changed between 2004 and 2017) only have 5% chance to rank in the top 10% percentile. The data show that these 70 files are at least twice likely to change compared to average files.

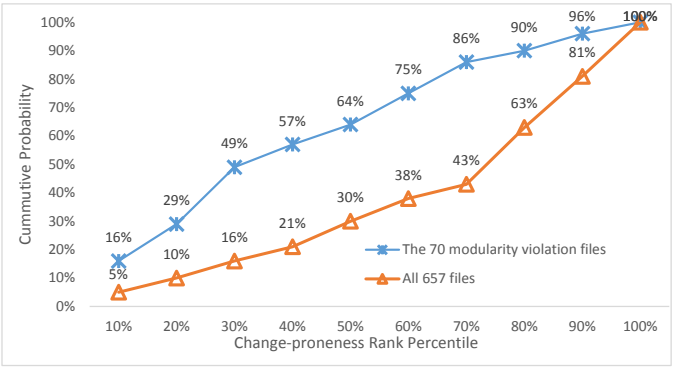


Fig. 4: Hardware-related and general change-proneness

Therefore, the hardware related concepts could be the main contributing factor to changes made to software modules.

V. Limitations and Future Work

In this paper, we treated source files as fine-grained modules. This is appropriate from the perspective of a low-level developer. However, we acknowledge that other stakeholders may view the system from different granularity levels. For example, product managers usually view modules as cohesive functional components to deliver the product value and competitiveness. This results in a very different definition of a module. From that perspective, a module may contain many source files, and the project owner may not be concerned with modularity violations among source files within a given module. In our future work, we plan to explore and investigate different criteria to decompose a system into modules and

conduct the modularity violation studies on other modular granularities.

Another limitation with this paper is that we only studied one project. The techniques and observations from in this study may not be directly generalized to other projects. We actually made some similar observations in another CPS called, MD PnP (Medical Device, Plug-and-Play), for medical care. MD PnP is also more modularized compared to the 129 traditional software systems we studied in the past. However, we found that the keywords identified for OpenWrt do not directly apply to MD PnP. The reason is that these two projects are in two completely different problem domains. OpenWrt is for supporting IoT. Therefore, the keywords are mostly related to network devices and sensors. However, MD PnP is in the medical domain. In our future work, we plan to conduct empirical study on a larger spectrum of cyber-physical systems and develop more general approaches to analyze modularity violations in systems from different domains.

Lastly, a remaining unresolved level of modularity violation is the hardware-to-hardware level modularity violations. This is largely challenged by the lack of systematic maintenance records on the hardware side. Unlike software components of a CPS, the revision history of the hardware parts are largely unavailable. This paper leverages software data to imply hardware related concepts. However, this approach cannot detect all the pure hardware level modularity violations. We will leave this challenge to future work.

VI. CONCLUSION

In this paper, we conducted a case study on an industry-scale cyber-physical system, OpenWrt, to investigate the feasibility of identifying and measuring modularity violations involving hardware components. To the best of our knowledge, this is the first work to address this problem in the area of CPS. What we found was that while the software architecture of OpenWrt is well-modularized, there were a number software modularity violations that appear to involve hardware-related concepts. Furthermore, we found that software components related to hardware were much more likely to change frequently. Combined, these findings suggest hardware components are a source of latent modularity violations in OpenWrt and demonstrate the feasibility of adapting methods to identify and measure modularity violations from pure software systems to cyber-physical systems. As part of our future work, we will investigate modularity violations in a broader spectrum of CPS systems.

ACKNOWLEDGMENTS

This material is based upon work supported, in whole or in part, by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract HQ0034-13-D-0004. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Department of Defense. The authors would like to acknowledge Joana Cardoso and Xiao Wang for assisting in the analysis of the OpenWrt data.

REFERENCES

[1] Mo, R., Cai, Y., Kazman, R., Xiao, L., Feng, Q. (2016, May). Decoupling level: a new metric for architectural maintenance complexity. In Proceedings of the 38th International Conference on Software Engineering (pp. 499-510). ACM.

[2] Baldwin, C. Y., Clark, K. B. (2000). Design rules: The power of modularity (Vol. 1). MIT press.

[3] Kruchten, P. B. (1995). The 4+ 1 view model of architecture. IEEE software, 12(6), 42-50.

[4] Gonzlez-Nalda, P., Etxeberria-Agiriano, I., Calvo, I. (2015, June). Flexible, modular, standard, free and affordable model for CPS control applied to mobile robotics. In Information Systems and Technologies (CISTI), 2015 10th Iberian Conference on (pp. 1-6). IEEE.

[5] MacCormack, A., Rusnak, J., Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science, 52(7), 1015-1030.

[6] Rajhans, A., Cheng, S. W., Schmerl, B., Garlan, D., Krogh, B. H., Agbi, C., Bhave, A. (2009). An architectural approach to the design and analysis of cyber-physical systems. Electronic Communications of the EASST, 21.

[7] Sinha, K. (2014). Structural complexity and its implications for design of cyber-physical systems (Doctoral dissertation, Massachusetts Institute of Technology).

[8] Baheti, R., Gill, H. (2011). Cyber-physical systems. The impact of control technology, 12, 161-166.

[9] Jantunen, E., Zurutuza, U., Ferreira, L. L., Varga, P. (2016, April). Optimising maintenance: What are the expectations for cyber physical systems. In Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC), 2016 3rd International Workshop on (pp. 53-58). IEEE.

[10] Derler, P., Lee, E. A., Vincentelli, A. S. (2012). Modeling cyberphysical systems. Proceedings of the IEEE, 100(1), 13-28.

[11] Cristalli, C., Boria, S., Massa, D., Lattanzi, L., Concettoni, E. (2016, October). A Cyber-Physical System approach for the design of a modular Smart Robotic Cell. In Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE (pp. 4845-4850). IEEE.

[12] Lee, E. A. (2015). The past, present and future of cyber-physical systems: A focus on models. Sensors, 15(3), 4837-4869.

[13] Lee, E. A. (2008, May). Cyber physical systems: Design challenges. In Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on (pp. 363-369). IEEE.

[14] Rajkumar, R. R., Lee, I., Sha, L., Stankovic, J. (2010, June). Cyber-physical systems: the next computing revolution. In Proceedings of the 47th Design Automation Conference (pp. 731-736). ACM.

[15] Wong, S., Cai, Y., Kim, M., Dalton, M. (2011, May). Detecting software modularity violations. In Proceedings of the 33rd International Conference on Software Engineering (pp. 411-420). ACM.

[16] Schwanke, R., Xiao, L., Cai, Y. (2013, May). Measuring architecture quality by structure plus history analysis. In Proceedings of the 2013 International Conference on Software Engineering (pp. 891-900). IEEE Press.

[17] Xiao, L., Cai, Y., Kazman, R. (2014, November). Titan: A toolset that connects software architecture with quality analysis. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 763-766). ACM.

[18] Mo, R., Cai, Y., Kazman, R., Xiao, L. (2015, May). Hotspot patterns: The formal definition and automatic detection of architecture smells. In Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on (pp. 51-60). IEEE.

[19] Malavolta, I., Muccini, H., Sharaf, M. (2015, September). A preliminary study on architecting cyber-physical systems. In Proceedings of the 2015 European Conference on Software Architecture Workshops (p. 20). ACM.

[20] Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q. (2016, May). Identifying and quantifying architectural debt. In Proceedings of the 38th International Conference on Software Engineering (pp. 488-498). ACM.