Effort Estimation for Architectural Refactoring to Introduce Module Isolation

Fatih Öztürk[1], Erdem Sarılı[1], Hasan Sözer[2], and Barış Aktemur[2]

[1] Vestel Electronics, Manisa, Turkey
{fatih.ozturk, erdem.sarili}@vestel.com.tr
[2] Department of Computer Science, Ozyegin University, Istanbul, Turkey
{hasan.sozer, baris.aktemur}@ozyegin.edu.tr

Abstract. The decomposition of software architecture into modular units is driven by both functional and quality concerns. Dependability and security are among quality concerns that require a software to be decomposed into separate units isolated from each other. However, it appears that this decomposition is usually not aligned with the decomposition based on functional concerns. As a result, introducing module isolation forced by quality attributes, while preserving the existing decomposition, is not trivial and requires a substantial refactoring effort. In this work, we introduce an approach and a toolset to predict this effort prior to refactoring activities. As such, a selection can be made among potential decomposition alternatives based on quantitative estimations. These estimations are obtained from scalable analysis of module dependencies based on a graph database and reusable query templates. We discuss our experiences and evaluate our approach on a code base used in a commercial Digital TV and Set-top Box software.

Keywords: software architecture; reverse engineering; refactoring; module isolation; effort estimation; dependability; security.

1 Introduction

Modularity is a key principle in software architecture design [12]. Decomposing the system into separate, modular units is driven by functional concerns and a set of relevant quality concerns such as dependability and security [2]. These quality concerns usually require that certain modules are decomposed and isolated from each other. For instance, distrusted modules must be isolated from the rest of the system to increase security. This is usually achieved by sandboxing [15] and placing each module into its own address space. Without such a fault isolation, errors can propagate among the modules of the system.

Isolation is usually supported by the operating system (e.g., process isolation [9]) or a middleware (e.g., encapsulation of Enterprise Java Bean objects [4]). Regardless of the underlying infrastructure, the application software architecture must be decomposed so that certain parts of the system can be quarantined. However, it appears that the required decomposition for module isolation

is usually not aligned with the decomposition based on functional concerns. The redesign and implementation of the whole system is likely to be an impractical approach for large-scale legacy systems. On the other hand, refactoring the existing systems is not trivial either; it requires that the interactions of a module with all the other parts of the system are captured and appropriately isolated [13].

In this work, we propose an approach and a toolset for predicting the refactoring effort for decomposition and implementation of software architecture for module isolation. As such, a selection can be made among potential decomposition alternatives based on quantitative estimations. In our approach, dependencies among the software modules are captured with a compiler frontend and stored in a graph database. These dependencies are queried based on a set of reusable query templates. Queries are instantiated according to the evaluated decomposition alternative. The novelty of our work is to facilitate the use of scalable and interactive architectural queries. We discuss our experiences in the application and evaluation of our approach by introducing module isolation to a set of modules taking part in a commercial Digital TV (DTV) and Set-top Box (STB) software architecture. We were able to estimate the required refactoring effort for a large code base with 85% accuracy on the average.

The remainder of this paper is organized as follows. Section 2 presents the industrial case study and a motivating example. We introduce our approach in Section 3. The evaluation of the approach is presented in Section 4. Related studies are summarized in Section 5. Finally, in Section 6, we provide our conclusions and discuss future work directions.

2 Industrial Case Study: DTV/STB Software

In this section, we introduce an industrial case study and a running example to be used in the rest of the paper. We investigated a software system being developed and maintained by Vestel[3], which manufactures DTV and STB systems. Conditional access (CA) system providers are among the customers of the company. These customers have various requirements that are subject to certification tests. One of these requirements is module isolation. Due to many different external interfaces such as USB and Ethernet, DTV and STB systems are exposed to an increasing number of dependability and security threats. Therefore, CA system providers require that certain modules of the system are isolated from each other by running them on different processes.

Vestel has a legacy code base that includes approximately 8M lines of code (LOC) in C/C++ excluding the chipset drivers (33M LOC including the drivers). The overall code base is composed of 4 layers: *i) Driver*: includes the platform-related software that is mostly in the kernel space; *ii) Platform Integration Layer*: provides abstraction for the functions provided by the Driver layer; *iii) Middleware*: implements the main business logic; *iv) Application*: implements the user interface. Module isolation requirements usually affect the 3^{rd} and the 4^{th} layers.

[3] http://www.vestel.com.tr

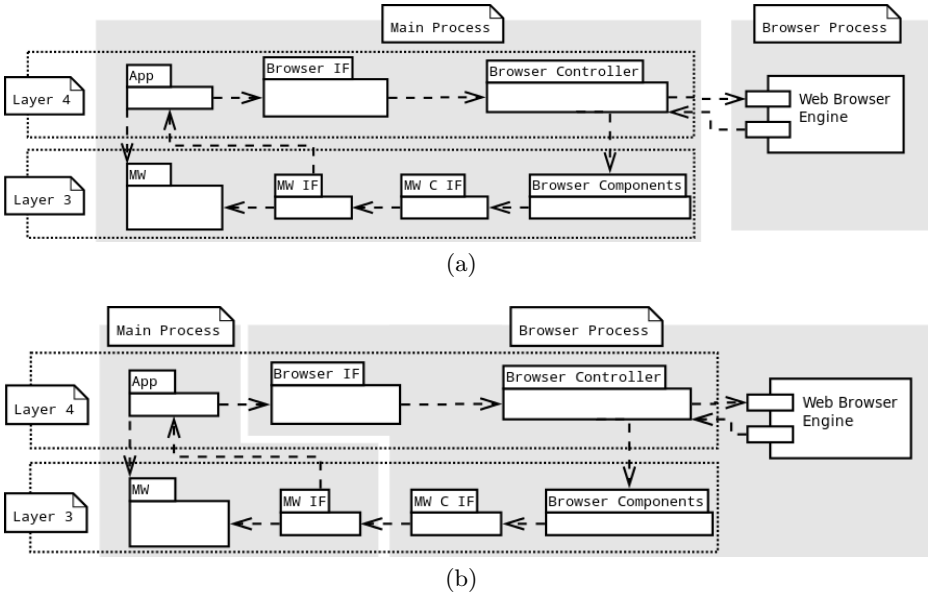

Fig. 1: Decomposition alternatives for the isolation of the web browser engine.

For instance, it was required by a CA system provider[4] that the web browser functionality should be isolated from the rest of the system. To satisfy this requirement, the corresponding module was planned to be isolated in a separate process as depicted in a module view of the software architecture in Figure 1(a). Refactoring a system for process isolation is not trivial for large code bases. It requires that the interactions of the isolated module(s) with all the other parts of the system are captured. All the function calls and direct accesses to shared data must be redirected through inter-process communication (IPC). As a result, additional glue layers and wrappers have to be developed [13].

There are usually many decomposition alternatives that satisfy a module isolation requirement. The implementation of these alternatives require different amounts of effort based on the module inter-dependencies. In fact, it was figured out later in the architectural refactoring phase that the decomposition depicted in Figure 1(b) was a better alternative in terms of effort. The development team abondoned the attempts to do decomposition given in Figure 1(a), resulting in wasted time and man-hours, and instead focused on Figure 1(b). In the following section, we introduce our approach for estimating the refactoring effort to evaluate various decomposition alternatives with automated and scalable analysis.

[4] Customer identity is undisclosed due to confidentiality agreements.

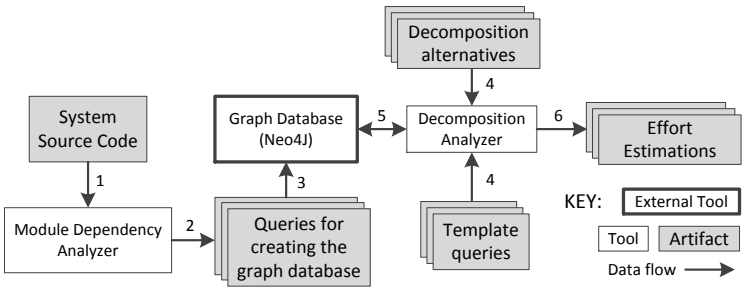

Fig. 2: The overall approach.

3 The Approach

The overall approach is depicted in Figure 2. First, a static code analysis, called the *Module Dependency Analyzer*, is applied to the system source code to identify module inter-dependencies. The analyzer is implemented as an LLVM [10] compiler pass that runs on intermediate level code. (Therefore, the pass is runnable on software written in any programming language provided that there is an LLVM front-end that translates the code to LLVM Intermediate Representation. In our case study, the code base is written in C/C++.) The output of the module dependency analyzer is a set of Cypher queries that build a graph database with Neo4J [8] (step 2). Then, these queries are executed to create a graph representation of all the identified module inter-dependencies (step 3). In our case, two modules shown in Figure 1 were analyzed. These modules are 20K LOC in total. The size of the generated queries was 76K LOC. It took around 3.5 hours to complete the execution of all the queries on a desktop computer. The graph had 25K nodes and 60K edges. A small, representative example is depicted in Figure 3. The graph database is built only once per code base. Then, it can be utilized many times to evaluate various decomposition alternatives (step 5). *Decomposition Analyzer* takes decomposition alternatives and template Cypher queries [8] as input (step 4). Each decomposition alternative specifies the set of modules that are separated from each other. Figure 1 depicts only the top level modules, each of which comprises many more modules. In our case study, we specified 10 module interfaces that are separated as a result of implementing the decomposition alternative depicted in Figure 1(b). Template queries are instantiated based on the evaluated decomposition alternatives. They also have coefficients to be adjusted based on the implementation. The execution of the queries outputs effort estimations (step 6). In our case study, executing the queries to evaluate 10 module interfaces took around 10 minutes.

We calculate the effort in terms of LOC to be written for glue layers and wrappers [13] required for realizing a decomposition alternative. These LOC mainly comprises IPC calls, callback handlers, and data (de)serialization imple-

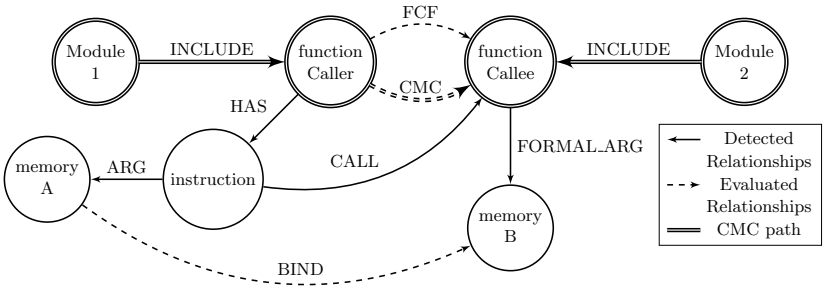

Fig. 3: Representation of module inter-dependencies as nodes and edges.

mented for coupled modules that are isolated in different processes. Hence, the effort is related to the amount of and the type of coupling among the isolated module interfaces. The queries that are instantiated for evaluation first detect function calls among such interfaces. Then, parameter bindings between formal and actual parametres are analyzed. For every call to be redirected, complexity of parameters and return value is calculated based on the use of pointers and nesting level of classes and structures. Finally, effort required for each cross-modular function call is summed up to represent total cost required for isolation of the given two modules. Figure 3 represents a simple parameter binding between the function *Caller* defined in *Module 1* and the function *Callee* defined in *Module 2*. Here *FCF* represents function call, *CMC* represents cross-modular function call, and *BIND* represents parameter binding between two memory locations.

A sample truncated query for evaluating module dependency is given in Listing 1. Hereby, *env.module2* and *env.module1* are parameters that define the separated modules (Line 5). The coefficient *SIMPLE_PARAM* (Line 9) defines the unit effort to handle a simple function argument. Nested argument structures are captured and the corresponding unit effort is calculated separately (Line 12).

The utilization of a graph database and reusable template queries provides scalability and genericity. Our analysis addresses the amount of coupling at the module view level. However, the approach can also be applied to different types of architectural views [5] by using appropriate set of template queries. In our case, the toolset can provide an effort estimation based on the decomposition structure and coefficients for unit costs. The coefficients can be adjusted based on the underlying isolation framework.

4 Evaluation and Discussion

We examined the implementation depicted in Figure 1(b) for the 10 module interfaces that are separated. We manually measured the real effort required for the realization of this decomposition in terms of *effective LOC* [6]. We also applied our approach on the previous version of the source code, before the

```
...
match (x:folder)-[:INCLUDE*1..]
    ->(caller:function)-[:FCF]->(callee:function)
    <-[:INCLUDE*1..]-(y:file) where
    x.name = env.module2 and y.name = env.module1
    create unique
    caller-[:CMC{param_point:0, return_point:0}]->callee;
...
match a-[r: BIND]->b set r.point = SIMPLE_PARAM;
match b<-[bind:BIND]-a-[LOAD*0..1]->()-[CAST*0..1]
    ->()-[:IS_A]->(strct) with a,b,bind,strct
    set bind.point = strct.point;
...
```

Listing 1: A truncated query template for dependency evaluation.

decomposition is implemented. We obtained estimations regarding the separation of the 10 module interfaces. We compared the estimated effort and measured effort in terms of the *relative error* [1] measure. Results are listed in Table 1. Estimations are 85% accurate per interface on the average. (We think that the per-interface average of relative error is a better indicator of accuracy than the relative error on the *overall effort*, which is much smaller: 1359 vs. 1306 $\Rightarrow$ 4% error.) In fact, if we do not consider the two exceptional interfaces, *H* and *I*, the accuracy is 91%. In the following, we discuss the reasons for estimation errors regarding these interfaces.

The measured effort is much less than the estimated effort for *Interface H*. This interface is generally composed of getter functions that return primitive C types. Serialization and extraction of the return values are identical for several

Interface	Measured Effort	Estimated Effort	Relative Error
A	128	133	0.04
B	94	78	0.17
C	175	189	0.08
D	102	88	0.14
E	80	66	0.17
F	321	302	0.06
G	65	68	0.05
H	165	211	0.28
I	125	70	0.44
J	104	101	0.03
Total	1359	1306	-
Average	-	-	0.15

Table 1: Comparison of measured and estimated effort.

functions. Therefore, such identical operations are implemented in a helper function, which reduce the effort to a large extent. On the other hand, the estimated effort is much less than the measured effort for *Interface I*, because this interface employs complex C structs with callback function pointers. The use of these callback functions are scattered among many modules. Hence, the isolation of *Interface I* required extra effort for transferring these functions through IPC.

5 Related Work

Vespucci tool [11] captures structural dependencies in multiple complementary views called *slices*. Each slice captures different types of dependencies to be analyzed separately. In this work, we capture all the dependencies in a graph database and query all types of dependencies regarding a certain part of the system, which is subject to refactoring for module isolation.

The FLORA framework [13] comprises a set of tools to estimate the performance overhead introduced by module isolation and optimize the software architecture decomposition [14]. The estimation is based on a dynamic analysis that collects statistics about the frequency of performed function calls and the data access profile of the system. In this work, we aim at estimating the maintenance effort for introducing module isolation. As such, we utilize static analysis. We also utilize a graph database and a declarative graph query language to achieve scalability [8].

Micro-kernel architectures [7] and operating systems with *sealed processes* [9] have been introduced for flexible multiprocessing support and better isolation to improve dependability and safety. To be able to exploit the multiprocessing support for isolation, the application software must be partitioned to be run on multiple processes. Our approach supports such a refactoring and predicts the re-engineering effort for making use of the multiprocessing support.

There have been also other approaches [3,4] to isolate software components from each other. However, they do not consider the restructuring and partitioning of legacy software to introduce this isolation.

6 Conclusion and Future Work

Module isolation can be necessary to satisfy several quality concerns. However, it appears that the required decomposition for module isolation is usually not aligned with the decomposition based on functional concerns. Therefore, the realization of this decomposition requires substantial maintenance effort. We have introduced an integrated toolset that predicts the refactoring effort to introduce module isolation by preserving the existing structure. We have illustrated our approach in the context of an industrial case study to introduce module isolation to a set of modules in a large code base. We obtained accurate estimations of the refactoring effort. As such, our approach proved to be practical for large-scale systems to support module isolation in software architectures.

As future work, we are planning to utilize our observations summarized in Section 4 to improve the accuracy of our estimations. We also plan to perform additional case studies.

Acknowledgements. We thank the software developers and managers at Vestel Electronics for sharing their code base with us and supporting our analysis.

References

1. Alsmadi, I., Nuser, M.: Evaluation of cost estimation metrics: Towards a unified terminology. Journal of Computing and Information Technology 21(1), 23–34 (2013)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
3. Buskens, R., Gonzalez, O.: Model-centric development of highly available software systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems IV, LNCS, vol. 4615, pp. 163–187. Springer (2007)
4. Candea, G., Fox, A.: Crash-only software. In: 9th Workshop on Hot Topics in Operating Systems (HotOS). pp. 67–72. USENIX Assoc., Berkeley, CA (2003)
5. Clements, P.C., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.A.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2 edn. (2010)
6. Fenton, N., Pfleeger, S.: Software Metrics: A Rigorous and Practical Approach. Thomson Learning Inc., 2 edn. (2002)
7. Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: Failure resilience for device drivers. In: 37th IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 41–50. Edinburgh, UK (2007)
8. Holzschuher, F., Peinl, R.: Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. In: EDBT/ICDT 2013 Workshops. pp. 195–204. ACM, New York, NY (2013)
9. Hunt, G., Aiken, M., Fähndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., Wobber, T.: Sealing OS processes to improve dependability and safety. SIGOPS Oper. Syst. Rev. 41(3), 341–354 (2007)
10. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Int. Symposium on Code Generation and Optimization (CGO). pp. 75–87. IEEE Computer Society, San Jose, CA, USA (2004)
11. Mitschke, R., Eichberg, M., Mezini, M., Garcia, A., Macia, I.: Modular specification and checking of structural dependencies. In: 12th Int. Conference on Aspect-oriented Software Development (AOSD). pp. 85–96. ACM, New York, NY (2013)
12. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
13. Sozer, H., Tekinerdogan, B., Aksit, M.: Flora: A framework for decomposing software architecture to introduce local recovery. Software: Practice and Experience 39(10), 869–889 (2009)
14. Sozer, H., Tekinerdogan, B., Aksit, M.: Optimizing decomposition of software architecture for local recovery. Software Quality Journal 21(2), 203–240 (2013)
15. Wahbe, R., Lucco, S., Anderson, T., Graham, S.: Efficient software-based fault isolation. SIGOPS Operating Systems Review 27(5), 203–216 (1993)