Operational Pattern Based Code Generation For Management Information System: An Industrial Case Study

[Anonymized for Blind Review]

***Abstract*—Code generation technology can significantly improve productivity and software quality. However, due to limited financial and human resources in most of small and medium software enterprises, there are many challenges when leveraging code generation approaches to large-scale software development. In this paper, an operational pattern based code generation approach is proposed for rapid development of domain-specific management information system. We demonstrate the approach with details: (I) semi-automatically extracting operational patterns from requirement documents, (II) building feature models to manage the commonalities and variability of each operational pattern, (III) mapping operational patterns into skeleton code with a template-based code generation technique, etc. Then we conduct an industrial case study in asset information management domain at CancoSoft Company for about 2 years, to analyze its feasibility and efficiency. 14 operational patterns are successfully extracted from 355 initial key phrases, and a code generator is implemented and applied to develop new Web applications. Preliminary findings show that the software development based on our approach yields a nearly 30% higher productivity as compared to traditional software development. Through code analysis, we find that around 70% of code can be automatically generated, and the generated code is also effective.**

Keywords—Operational Pattern, Code Generation, Software Product Line, Domain Engineering, Exploratory Case Study.

Category: other (software reuse)

I. INTRODUCTION

Code generation can automate source code creation through generic frames, classes, prototypes, templates, aspects, and code generators to improve programmers' productivity [1]. It is an effective method to achieve software reuse in application development and has many successful cases in embedded software [2], web applications [3] and distributed client-server systems [4].

There are four common approaches to using code generation technology in large-scale, industrialized software development. However, due to limited financial and human resources in most Small and Medium Software Enterprises (SMSEs), there are many challenges for these approaches. The details are listed as following:

(1) Model-Driven Development (MDD) focuses on code generation from models and the executability of models [5]. MDD defines the Platform-Independent Model (PIM) at a high level of abstraction and then defines rules to transform the PIM to Platform Specific Models (PSMs). Finally, code in some third generation languages (e.g., C++, Java) is automatically generated from PSMs. Some transformation tools are available as RSA, AndroMDA, ECO II, EMF, etc. However, extensive handcrafting of implementations is required for multiple levels of abstraction, detailed design models, and automated support for transforming and analyzing models, which may cause accidental complexities that make software development difficult and costly[6][7].

(2) Domain Specific Language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain [8]. With tool support, the code can be directly generated from a high-level abstract description defined in DSL. The cost of designing, implementing and maintaining a DSL, as well as training for DSL users is high. In addition, there are difficulties in integrating DSL with other components in a software system [10].

(3) Software Product Line Engineering (SPLE) is a paradigm to develop software applications using platforms and mass customization [11]. In domain engineering process of SPLE, commonalities and variability of a product line are defined and realized. In application engineering process of SPLE, applications of a product line are built by reusing domain artifacts and exploiting configurable variability. Code generation approaches, including generative programming [4] and two approaches mentioned above, can be applied to reuse software artifacts to quickly build new applications in SPLE. However, in an SPLE approach, it takes nearly 444 person months for a typical product line supposing that each product size is 100-kilo source lines of code (KLOC) [12]. Taking into account the urgent market need for a product, the introduction of product lines will cause a long delay to delivery time.

(4) Program synthesis is the task of automatically synthesizing a program in some underlying language from a given specification using some search technique [13]. It can aid in automated debugging and, in general, leaves the human programmer free to deal with a high-level design of the system. Additionally, synthesis can discover new non-trivial programs that are difficult for programmers to build [14]. Microsoft Research Redmond lab has made some achievements in the program synthesis. A preliminary summary of issues and methods are discussed in [14][15]. But the program synthesis is mainly used in the generation of new algorithms, and it is still in its infancy for generating application code.

Just in this content, this paper is motivated by the need to provide a feasible approach for SMSEs to adopt code generation technology on software development with a higher productivity and no adverse effect on quality. In our approach, the code is generated from templates which are developed from legacy software artifacts, while the code is directly generated from well-designed models in MDD. We focus more on specific domains and application code generation, which makes our approach different from program synthesis. The metadata of

configuration is defined in eXtensible Markup Language (XML), which avoids the challenges of DSL. Our approach consists of domain engineering and application engineering, which are tailored from SPLE. But we shrink the variability management on only one aspect that is the operational pattern. The case study suggests that the simplification to SPLE is feasible in the software development of asset information management domain. In summary, we have made the following two contributions:

(1) We propose an Operational Pattern based Code Generation (OPCG) approach for SMSEs in information management domain. In essence, diverse methods from MDD, DSL, SPLE and Natural Language Processing (NLP) are integrated and tailored in the OPCG approach to maximize the development productivity and minimize negative effort on project quality.

(2) We report on the design and execution of a case study in asset information management domain. In our study, we implement our approach and practically apply it to new Web application development. Data are collected from 5 projects at Shanghai Canco Software Co., Ltd. (CancoSoft). Result analysis show that: first, the productivity of the OPCG-based software development is improved by almost 30% compared to traditional software development, and defect density is reduced. Second, around 70% of code can be automatically generated.

The rest of this paper is structured as follows. Section II describes the OPCG approach, including semi-automatic extraction of operational patterns and template-based code generation. Then in section III, we describe the design and execution of our field study. After analyzing the data collected from projects, we provide answers to our RQs and have a discussion on our work in section IV and V. Conclusions and future work are discussed at the end of this paper.

II. Operational Pattern Based Code Generation Approach

In this section, we first give the definition of operational pattern and then propose a novel domain-specific code generation approach based on OPs. Two key technologies in the approach are described in details, namely extraction and modeling of OPs, and template-based code generation.

TABLE I. Anatomy Table of Operational Pattern

Attributes	Explanation
Name	Name of OP.
Version	Version number.
Date	Date of the nearest modification.
Author	Author or organization that fill this table.
Description	Describe what the OP is.
Keywords	Representative keywords or tags related to this OP. They can help match this OP to specific operations.
Dependencies	Other OPs that this OP depends on.
Constraints	Describe internal or external constraints.
Pre-Condition	Conditions before applying this OP.
Post-Condition	Conditions or results after applying this OP.
Sequences	A sequence of operations that this OP consists of.
Common Aspects	Common aspects of each operation will be emphasized here.
Variation Aspects	Variable aspects of each operation are extracted here. They can be optional operations or internal variability of an operation.
Known Uses	Uses (or functions) are known as containing this OP.
Comments	Other complementary description.

A. Operational Pattern

As a specific software requirement pattern [16][17], Operational Pattern (OP) is a reusable sequence of operations that frequently appear in a series of Software Requirements Specifications (SRSs). By transforming the operational patterns to code directly, the coding effort on implementing repeated operations can also be avoided in the software development of a specific domain. An OP contains 15 attributes as shown in Table I.

B. OPCG

Based on operational patterns, we propose a novel domain-specific code generation approach named OPCG approach, as shown in Fig. 1. It adopts domain-oriented software development method and consists of two phases: domain engineering phase for reuse and application engineering phase with reuse.

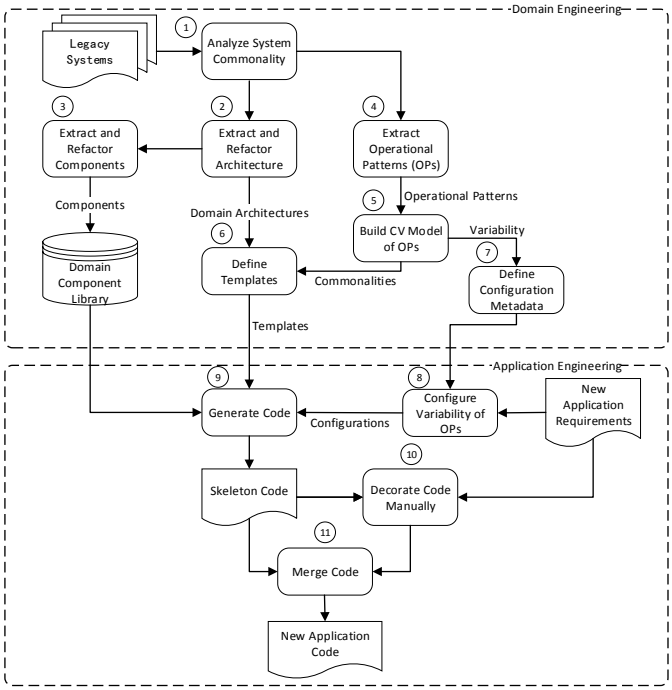

Fig. 1. The OPCG Approach

The domain engineering phase involves identifying commonalities and differences between legacy systems in a domain and implementing a set of shared software artifacts [18].

[Step 1] System commonality is analyzed to find reusable parts from legacy software artifacts, including documents (SRSs, design documents, user manual, etc.) and code.

[Step 2] Domain architecture are extracted and refactored. Architects first have to identify key requirements that have an essential impact on the architecture, and then create a conceptual architecture before building structural models of the software [11]. Structural models decompose legacy software systems into parts and relationships with some architecture views, such as development view and process view. To get a reference architecture that captures a high-level design for all applications, architects need to refactor the domain architecture extracted

from legacy systems, adopting architecture style, design pattern, and new technologies.

[Step 3] Components are extracted and refactored following Step 2. Some important tasks of extracting components are to identify reusable modules and to extract common parts for refactoring high-quality components. Component refactoring involves component redesign, in which dependency of the component configuration in a specific legacy application is removed. In addition, variable interfaces and implementation of a component are required to meet various application requirements.

[Step 4 and Step 5] OPs are semi-automatically extracted and feature models are built on Commonalities and Variability (CV) of OPs. Details are explained in the following subsection C.

[Step 6] Templates are defined according to commonalities of OPs and domain architecture. A template is "an output document with embedded actions which are evaluated when rendering the template" [21]. The fixed part of the template is copied to the output without modification while placeholders are replaced with actions or expressions when processing templates.

[Step 7] Metadata of configuration is defined according to variabilities of OPs. The metadata is a collection of elements and attributes to configure the variability into a series of configuration files based on XML. The difference between templates and configurations is: templates contain all possible combinations of code, while configurations are to make a valid combination of code according to a specific requirement.

In the application engineering phase, applications are efficiently developed by reusing software artifacts prepared in the domain engineering phase.

[Step 8] The variability of OPs is configured upon new application requirements with the metadata of configurations.

[Step 9] Code is generated with the inputs of domain components, templates, and configurations. In order to apply the commonalities and variability of each OP into concrete code with high efficiency, a template-based code generation strategy is needed. The output is skeleton code, which consists of the code fragments mapping to the architecture, components, and definite operations. The template-based code generation technology is explained in subsection D.

[Step 10] The skeleton code lacks specific business logic code, so developers need to decorate the skeleton code manually, including adding new code and modifying generated code.

[Step 11] Requirements change frequently and part of the skeleton code needs to be regenerated. New application code is obtained by merging the regenerated skeleton code and the decorated code (for several times).

In the next two subsections, extraction and CV modeling of OPs, and template-based code generation are explained in details.

C. Extraction and CV Modeling of OPs

The extraction of OPs is divided into three steps. After that, OPs are specified with feature models through the commonality and variability analysis.

1) *Identify Key Phrases*: Requirement documents are usually non-structural and written by different authors in natural language. In order to identify key phrases from requirements automatically, we leverage some natural language processing techniques combined with some simple but effective heuristic rules to design our extraction algorithm. There are three main parts in our extractor, namely *dependency analyzer*, *key phrase extractor* and *key phrase filter*.

Dependency analyzer provides a representation of grammatical relations between words in a sentence. A dependency analyzer is implemented by using Stanford Parser. Based on the dependencies, we design a rule-based algorithm to extract phrases in a sentence. A phrase can be represented as two-tuples (p, o), where p indicates the predicate and o indicates the object. The predicate of a sentence can be captured by *root* (more details about this symbol can be found in [19]) and extracted directly from dependencies. For each dependency, we first check whether the governor of this dependency is the root node. Then, if the relation name is *dobj* (means a direct object) or *nsubjpass* (means a passive nominal subject), the dependent of this dependency is right the object (See Algorithm 1). Since the noun phrase in a sentence will be split by *nn* relation (*nn*(*record-4, repair-3*)), a Merge operation will be carried to construct the whole name of the object. After this step, all the candidate phases are obtained. For example, the phrase extracted from "*Add a repair record in device management system*" will be *(add, repair record).*

In key phrase filter, $Support$ is introduced to measure the commonness of key phrases and is defined as follows.

$$Support_{kp} = \frac{\mathrm{ND(kp)}}{\mathrm{n}} \quad (1)$$

Where n is the total number of documents, and $ND(kp)$ is the number of documents which contain key phrase kp. Note that only common phrases (i.e. the value of $Support$ is not less than a certain number (θ) between 0 and 1), will be treated as key phrases.

Input: TypedDependencies *tdls* generated from Dependency Analyser.
Output: Candidate phrases.
1. *root* ← getRootNode(*tdls*)
2. **for each** *td* ∈ *tdls* **do**
3. *gov* ← getGovernor(*td*)
4. *dep* ← getDependent(*td*)
5. *reln* ← getRelationName(*td*)
6. **if** *gov* == *root* **then**
7. **if** *reln* == 'nsubjpass' **or** 'dobj' **then**
8. *object* ← *dep*
9. **end if**
10. **end if**
11. **end for**
12. MergeNN(*tdls*, *object*)
13. **return** (*root*, *object*)

Algorithm 1: Identify Candidate Phrases

Not all key phrases can be identified by using this method, and not all phrases identified are really indispensable to define operational patterns. However, it does assist domain experts to identify key phrases easier.

2) *Refine Phrases*: Two kinds of generalization, i.e., verb generalization and entity generalization, are performed to refine the initial key phrases. The verb generalization merges two or more different phrases with similar operations into one phrase by abstracting verbs. And the entity generalization does merge by abstracting entities. The differences among verbs or entities are categorized into the variability of OP. Through the two generalizations, an OP is defined in a more general form, which can help classify common operations among systems and increase OP's reusability in application development.

3) *Extract Operational Patterns*: OPs are manually extracted from refined phrases. An OP is generic and regarded as a higher abstract representation than a phrase. Accordingly we semantically categorize all the refined phrases with some factors, such as the entities that a phrase refers to, the complexity of the operation in a phrase, and the external or internal constraints. The process of extracting appropriate OPs involves a lot of discussion with experts from both of the domain engineering and the specific domain. Each OP is specified with domain terms and constraints.

4) *Build CV Model of OPs*: To further analyze all CVs of an OP, an "Operation-Function" (Oper-Func) matrix is established. Rows of the matrix represent operations, while columns represent functions (or requirements) which contain those operations appeared in rows. The value of a cell is chosen from "Mandatory", "Optional" and "Exclude". If an operation is "mandatory" for all functions, it is a commonality; otherwise, it is a variability (with constraints if any). After that, feature model [20] is used to manage the CVs of each OP. Finally, the metadata of configuration is defined according to the variability. Table II shows variants, variation points and configuration metadata of "Approve Request". The variation points are identified from four type of variant. The content surrounded by brace may be configured for more than once.

TABLE II. AN EXAMPLE OF DEFINING THE METADATA OF CONFIGURATION

Variant	Variation point	Metadata of configuration
Basic Information	Function name	Element: functionName
	Approval level, approver	Attribute: level, role
Data entity	Database table	Element: requestTableName
Optional operation	Whether automatic remind, reminder type	Element: prompt Attribute: active, type
	Query approval records	Element: optionalOper Note: Other metadata is contained in "Query Multi-table Information" OP.
Fields and user interface	Request form: Title, field, label name, input widget type, widget length, required filling, auto filling, visible in all of approval	Element: requestView Attribute: title, {field, labelName, widgetType, length, required, autoFill, visible}
	History record: Title, field, label name, output widget type, widget length, visible	Element: recordView Attribute: title, {field, labelName, widgetType, length, visible}
	Approval result: Title, field, label name, input widget type, widget length, required filling, auto filling, visible in all of approval	Element: approvalView Attribute: title, {field, labelName, widgetType, length, required, autoFill, visible}

D. Template-based Code Generation

OPCG is a template-based code generation approach. There are four common approaches (i.e., abstract syntax trees based, print statements based, term rewriting and template-based) to implementing a heterogeneous generator [21]. Particularly, the template-based approach can be used for generating all kinds of unstructured text. Additionally, a good template can greatly improve the efficiency of running code as well as maintenance.

A technical framework of the template-based code generation is designed, as illustrated in Fig. 2. Specific requirements on OPs are analyzed from new application requirements. These requirements will be converted into the elements and attributes in the XML-based configuration files. Commonalities of OPs are included in the template files while variable parts are included in the configuration files, which are parsed by a configuration parser. Outputs of the parser are taken to replace placeholders in the template files with actions or expressions through a template engine. The generated code are program files in text, such as HTML files and Java files.

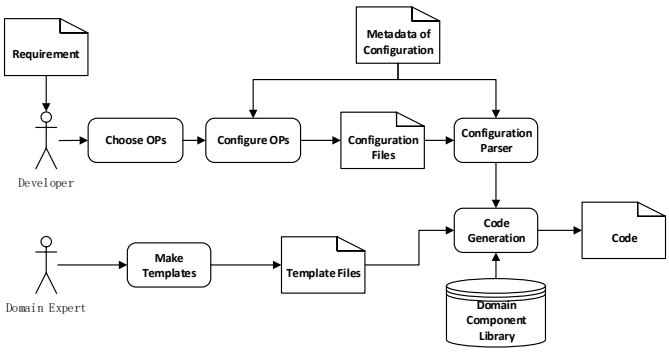

Fig. 2. Technical Framework of Template-based Code Generation

III. CASE DESIGN AND EXECUTION

Our case study aims to address three Research Questions (RQs):

RQ1: Is the operational pattern based code generation approach feasible in management information system development?

RQ2: Does the OPCG-based software development yield a higher productivity than traditional software development? How is its effect on software quality?

RQ3: What is the coverage of the generated code to total code in new application projects? Is the generated code effective?

A. Case Description

The case study is conducted at CancoSoft for about 2 years, using some frequently used exploratory case study methods (c.f. [22], [23]). CancoSoft is a software services provider that focuses on asset information management systems. In recent years, due to costly maintenance and repetitive coding effort, CancoSoft suffers a software development with low efficiency. In this context, the OPCG approach is proposed and a code generator named CodeGen is developed. CodeGen has been

applied to the practical software development at CancoSoft for almost one and a half years.

B. Case Design

The case study is divided into two periods according to our approach. In the first period, we aim at the domain engineering, or more specifically, the preparation of reusable software artifacts, including domain components and templates, as well as a code generator. In the second period, we apply the OPCG approach to the software development at CancoSoft and collect relevant data on development in the meantime. Further analysis of those data is performed to answer all RQs.

Particularly, in order to answer the RQ2, we refer to the case design in [23], which aims at evaluating the effectiveness of model-based software development on the productivity of enhancement tasks by analyzing statistics on software size, total effort and rework effort. But our work differs from theirs mainly in following two aspects: (1) The unit of analysis for our case study is a project rather than a task. So we need to reconsider the criteria for calculating software size and total effort, etc. (2) Different data analysis methods should be used as the statistical data has changed.

Software size: Given that all projects are developed for Web applications in our study, we calculate the software size of each project on two metrics: Lines of Code (LOC) and total number of Web pages (totPage). The totPage is the sum of total number of new Web pages and Web pages given by the customer [24].

Maintenance effort (MaintEffort): The maintenance work, such as correcting faults and improving performance, mainly concentrates in three months since delivery. Part of maintenance effort is collected from historic record.

Defect density: The number of defects per KLOC. Those defects are found during system test and maintenance phase. We count them according to four severity level, i.e., minor, moderate, major and critical, corresponding to the coefficient of 0.25, 1, 2 and 3 respectively.

Total effort (Effort): Total effort spent on the project includes the effort from requirements phase to the maintenance phase. Total effort and maintenance effort are both measured in person days.

Code coverage: To obtain the coverage of the generated skeleton code to total code, we collect data on LOC and totPage of this code. More code analysis is carried to check the effectiveness of the skeleton code.

C. Implementation of OPCG

It takes six months to implement OPCG with the effort of experts and developers. We analyze the domain commonalities with documents and code collected from 11 legacy systems. When extracting and refactoring the domain architecture and components, we emphasize several key activities here as a supplementary for the activities mentioned in the overall approach. Domain architecture are modeled in 4+1 views [25], and refactored by introducing abstraction hierarchies, removing unnecessary abstractions, breaking dependency cycles [26], and using Spring Framework2,. When extracting components, we construct component diagrams to get a clear picture for how components are wired together and choose reusable components with appropriate granularity.

Following the process of extracting OPs described in Section II-C, we successfully extract 14 OPs from SRS documents of 11 legacy systems. 355 key phrases are identified with our automatic extractor. These key phrases are then refined with both of verb generalization and entity generalization. E.g., for verb generalization of “Upload attachment” and “Download attachment”, which have similar functionalities, they can be merged into “Manage attachment”. “Query asset information” and “Query system log” are generalized into ”Query objects”. Refining produces 15 new phrases, but the total number of phrases decreases by 274 after that. Through lots of discussion with three specific domain experts and the domain engineering expert, 14 OPs are extracted from 96 key phrases. They are “C.R.U.D.(Create, Retrieve, Update, Delete) single table information”, “C.R.U.D. multiple table information”, “Approve request”, “Prompt and alert”, ”Report generation”, “Multi-dimensional analysis”, “Check inventory” and “Split asset”. We develop the specification for each OP with constraints and domain terms, then build feature models for the CVs analyzed from “Oper-Func” matrixes. Fig. 3 is the feature model of “Approve request”.

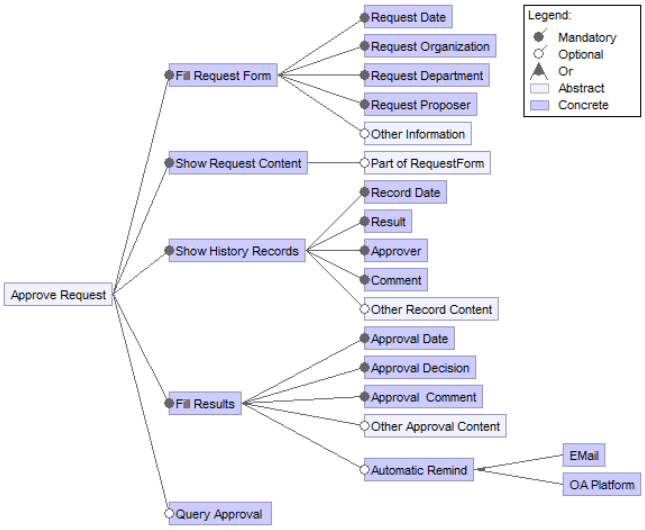

Fig. 3. The Feature Model of an Operational Pattern

CodeGen is a template-based system with a configuration parser and a code generator. Most code in templates are developed with experienced developers and optimized for many times to ensure the generated skeleton code is error-prone. We implement CodeGen with an open source template engine called Freemarker, which can generate all kinds of unstructured text output from templates. In addition to predefined directives, Freemarker also provides a mechanism for user-defined directives. We use this mechanism to define metadata in configuration along with transformational rules in CodeGen. These rules are included in a configuration parser which parses the configuration data from XML-based files. The outputs of CodeGen are primary files to build a web application, such as HTML files, Java files and Java Server Pages (JSP).

When applying OPCG to new application development, developers need to analyze new application requirements and determine specific configurations for the variability of each OP. Feature models can help developers compose a valid configuration. The generated skeleton code is not executable and need to be decorated with more logic code. When requirements change, new skeleton code is generated and needs to be merged with the decorated code. Though CodeGen has provided a service to locate and display the differences between two files, the merging need to be done by developers manually for its complexity.

D. Data Collection

We collect data from 5 projects which have similar software scale. 3 previous projects are developed with object-oriented development, which is treated as the traditional software development. After about one week's CodeGen training for developers, the tool is applied to the development of two new but typical Web applications. Data on software size, total effort, maintenance effort, defects density and code coverage are collected and then analyzed to find the effectiveness of the OPCG-based software development and the coverage of skeleton code to total code. Table III demonstrates the development information of those projects, where project 4 and 5 are developed with the OPCG approach.

TABLE III. DEVELOPMENT INFORMATION OF PROJECTS

Project	Industry Sector	Team Size	Dev. Effort (man-day)	totPage	KLOC	Defects	MaintEffort (man-day)
1	Bank	6	710	138	129.	666	50
2	Bank	5	650	120	117.	682	52
3	Bank	6	680	129	121.	709	56
4	Bank	5	500	119	115.	407	38
5	Bank	4	430	111	102.	330	35

IV. ANALYSIS AND RESULTS

RQ1. In most of Management Information System (MIS), there exist lots of repeated operations, such as like C.R.U.D., approving request, exporting report, etc. These operations can be detected as patterns during the requirements phase. We proposes a systematic approach to extract and reuse OPs in a explicit way. Additionally, we have implemented our approach in asset information management domain, and the CodeGen tool has been successfully applied to develop new applications. The effort spent on the implementation of our approach was about 600 man-day, which was acceptable for the company and would be compensated with further usage of CodeGen. These show that OPCG is feasible in the software development of MIS.

RQ2. The productivity for each project can be calculated by software size of project to the total effort spent on project [24]. Given that all selected projects have a similar scale on team size and development period, we divide them into group A, which consists of 3 previous projects, and group B, which consists of 2 new projects. Fig. 4 illustrates the obvious productivity difference between two groups on (totPage/Effort) and (KLOC/Effort) respectively. With the statistical results in Table III, we can calculate the two percentages of productivity improvement. One percentage is calculated as (0.25-0.19)/0.19*100%=31.58% with two means of (totPage/Effort) A and (totPage/Effort) B. The other is 27.78% with two means of (KLOC/Effort) A and (KLOC/Effort) B. The maintenance effort of each project is normalized with the total effort of respective project, which shown in Fig. 5. The deviation between (MaintEffort/Effort) A and (MaintEffort/Effort) B is negligible, which suggest that the OPCG-based development has no significant effect on the maintenance of new projects. Fig. 6 shows the defect density of five projects. The average density decreases by 2.2 (Defects/KLOC) after adopting the OPCG approach.

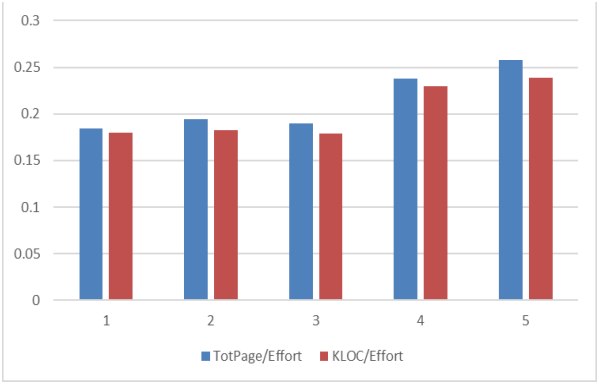

Fig. 4. Productivity on totPage and KLOC

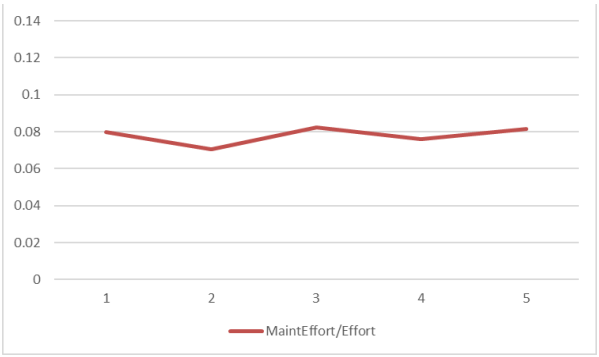

Fig. 5. Proportion of Maintenance Effort to the Total Effort

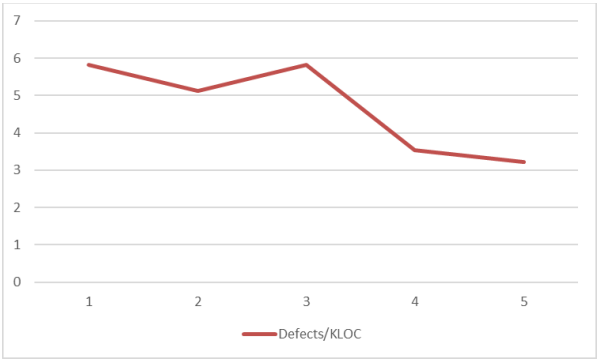

Fig. 6. Defect Density (the Number of Defects per KLOC)

RQ3. Table IV shows the statistical data of the skeleton (Skel.) code and final code in 2 new projects. Comments and blank lines are exclusive when counting valid code, while all of them are summed in total code. The two average ratios are 73.13% on valid code and 73.36% on total code respectively. We find that the coverage ratio has a negative correlation to the software

size of project. The reason is that the amount of templates is limited and much more code need to be decorated in a larger project. In addition, we compare the differences between the original files generated by CodeGen to the final delivered files. All original files are kept in the final files, and the average percentage of useless code (i.e., the code deleted in the final files) in skeleton code is less than 10%. Comparing to the common code coverage which ranges from 40% to 90% [28][29], our code coverage is above average. We preliminarily conclude that the skeleton code is effective and the average coverage is nearly 70% to the total code.

TABLE IV. STATISTICS OF CODE COVERAGE

Project	*(1)Valid Skel. Code*	*(2)Valid Final Code*	*(3)Total Skel. Code*	*(4)Total Final Code*	*Ratio of (1)/(2)*	*Ratio of (3)/(4)*
4	61,473	87,416	81,421	115,021	70.32%	70.79%
5	56,964	77,078	78,032	102,770	73.90%	75.93%

V. DISCUSSION

In Section II-C1, we propose a rule-based approach to extract key phrases from requirement documents automatically. The approach works well by leveraging the dependency analysis which is widely used in Natural Language Processing area. Note that our key phrase extraction algorithm is quite robust in the projects of asset information management domain, this is mainly because we use the dependency analysis to have a deep understanding of sentences. That is to say, even there are many modifiers before a real object, or the given sentence is in the past tense, our algorithm can still extract the key phrase accurately. However, since the approach is rule-based, it is obvious that many important key phrases could be missed when applying on a more complex requirements document. It is interesting to design a bootstrapping algorithm which can learn the key phrase extraction rules iteratively with only a few seeds. We assign 1 to the θ for the support in the case study. The reason why we choose the value is that the operational pattern we want to extract is a higher abstract representation from phrases. It is supposed to be generic for all members, with no exception, in a category of phrases.

There are probably two threats to validity of our experiment:

(1) No two teams with similar experience level can be assigned to develop the same project with traditional and the OPCG approaches respectively, due to limited resources. We believe the above concern is not necessary. Results of our experiment are relatively fair, as complexity of 3 previous projects and 2 new projects is nearly the same.

(2) Productivity gain might solely be attributed to the knowledge and skill improvement of our developers. We believe the above point is not valid, as we cannot observe productivity gain in other new projects with traditional approach at CancoSoft.

Though our approach has been applied for only one domain since proposed, we believe that it can play a good role in other operation-intensive domains with well-formatted requirement documents, such as automobile industry. The NLP techniques can help experts extract OPs with efficiency and less omissions. The code generation can reduce the tedious and time-consuming coding effort on repeated operations among projects. It is even possible to generate most of program with configurations being defined by staff with less programming skill.

VI. CONCLUSION AND FUTURE WORK

We present an operational pattern based code generation approach for small and medium software enterprises in the development of management information system. Operational pattern proposed in this paper is actually reusable sequence of operations in system requirements. NLP techniques are leveraged in extracting operational patterns, and feature model is adopted to manage the commonalities and variability of each operational pattern. Skeleton code can be generated from OPs through a template-base code generation technique. We also explain implementation and application of the approach with an industrial case study in the asset information management domain. In this study, we analyze the feasibility of our approach for the software development in the asset information management domain. Effect of development productivity and project quality is studied by comparison between 2 new projects developed with our approach and 3 previous projects with traditional approach. It shows that the productivity of our approach is improved by almost 30% compared to the productivity of traditional software development, with positive effect on quality. Furthermore, preliminary findings on the code of new projects suggest that generated code is effective and around 70% of code can be automatically generated with our code generation tool.

In essence, to maximize the development productivity and minimize negative effort on project quality, diverse methods from NLP, MDD, DSL and SPLE are integrated and tailored in the OPCG approach.

Future work: More metrics, besides LOC and totPage, might be needed to measure software productivity, especially when this approach is applied in other business domains. Feasibility of our approach should be further verified with more business domains. In extracting operational patterns, we will design a bootstrapping algorithm which can handle more complex requirement documents, and execute two generalizations with automatic methods. Generating templates from operational pattern models is a further research area we are interested in.

ACKNOWLEDGMENT

This research is supported by 973 Program in China (Grant No. 2015CB352203), National Natural Science Foundation of China (Grant No. 61472242), and Key Lab of Information Network Security, Ministry of Public Security (Grant No. C14609).

REFERENCES

[1] A. S. Abbas, W. Jeberson, and V. Klinsega, "A literature review and classification of selected software engineering researches," International Journal of Engineering and Technology, vol. 2, no. 7, p. 1, 2012.

[2] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, "Generative programming for embedded software: An industrial experience report," in Generative Programming and Component Engineering. Springer, 2002, pp. 156–172.

[3] A. Bozzon, S. Comai, P. Fraternali, and G. T. Carughi, "Conceptual modeling and code generation for rich internet applications," in

Proceedings of the 6th international conference on Web engineering. ACM, 2006, pp. 353–360.

[4] K. Czarnecki and U. W. Eisenecker, “Generative programming,” Edited by G. Goos, J. Hartmanis, and J. van Leeuwen, p. 15, 2000.

[5] A. MacDonald, D. Russell, and B. Atchison, “Model-driven development within a legacy system: an industry experience report,” in Software Engineering Conference, 2005. Proceedings. 2005 Australian. IEEE, 2005, pp. 14–22.

[6] A. G. Kleppe, J. B. Warmer, and W. Bast, MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional, 2003.

[7] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in 2007 Future of Software Engineering. IEEE Computer Society, 2007, pp. 37–54.

[8] A. Van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography.” Sigplan Notices, vol. 35, no. 6, pp. 26–36, 2000.

[9] M. Fowler, Domain-specific languages. Pearson Education, 2010.

[10] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” ACM computing surveys (CSUR), vol. 37, no. 4, pp. 316–344, 2005.

[11] K. Pohl, G. Bo ̈ckle, and F. J. van der Linden, Software product line engineering: foundations, principles and techniques. Springer Science & Business Media, 2005.

[12] B. Boehm, R. Madachy, Y. Yang et al., “A software product line life cycle cost estimation model,” in Empirical Software Engineering, 2004. ISESE’04. Proceedings. 2004 International Symposium on. IEEE, 2004, pp. 156–164.

[13] S. Gulwani, “Dimensions in program synthesis,” in Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming. ACM, 2010, pp. 13–24.

[14] S. Srivastava, S. Gulwani, and J. S. Foster, “From program verification to program synthesis,” in ACM Sigplan Notices, vol. 45, no. 1. ACM, 2010, pp. 313–326.

[15] S. Gulwani, “Synthesis from examples: Interaction models and algorithms,” in Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on. IEEE, 2012, pp. 8–14.

[16] S. Renault, O ´. Me ́ndez Bonilla, J. Franch Gutie ́rrez, M. C. Quer Bosor et al., “A pattern-based method for building requirements documents in call-for-tender processes,” 2009.

[17] S. Al-Fedaghi and M. Almutairy, “Toward conceptual representation of patterns,” in 2015 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015, pp. 1–8.

[18] S. Deelstra, M. Sinnema, and J. Bosch, “Product derivation in software product families: a case study,” Journal of Systems and Software, vol. 74, no. 2, pp. 173–194, 2005.

[19] M.-C. De Marneffe and C. D. Manning, “Stanford typed dependencies manual,” URL http://nlp.stanford.edu/software/dependencies manual.pdf, 2008.

[20] D. Beuche, H. Papajewski, and W. Schro ̈der-Preikschat, “Variability management with feature models,” Science of Computer Programming, vol. 53, no. 3, pp. 333–352, 2004.

[21] J. Arnoldus, M. Van den Brand, A. Serebrenik, and J. J. Brunekreef, Code generation with templates. Springer Science & Business Media, 2012, vol. 1.

[22] P. Runeson and M. Ho ̈st, “Guidelines for conducting and reporting case study research in software engineering,” Empirical software engineering, vol. 14, no. 2, pp. 131–164, 2009.

[23] D. Kamma and S. K. G, “Effect of model based software development on productivity of enhancement tasks - an industrial study,” in 21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers, 2014, pp. 71–77. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2014.20

[24] B. Kitchenham and E. Mendes, “Software productivity measurement using multiple size measures,” Software Engineering, IEEE Transactions on, vol. 30, no. 12, pp. 1023–1035, 2004.

[25] P. B. Kruchten, “The 4+ 1 view model of architecture,” Software, IEEE, vol. 12, no. 6, pp. 42–50, 1995.

[26] M. Stal, “Software architecture refactoring,” in Tutorial, in The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), 2007.

[27] P. Mohagheghi and V. Dehlen, “Where is the proof? - a review of experiences from applying mde in industry.” Lecture Notes in Computer Science, vol. 5095, pp. 432–443, 2008.

[28] MODELWARE D5.3-2 Enabler ROI, Assessment, and Feedback. Revision 1.1 (2006), http://www.modelware-ist.org

[29] D. Lucredio, E. S. D. Almeida, and R. P. M. Fortes, “An investigation on the impact of mde on software reuse,” in Proceedings of the 2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse, 2012, pp. 101–110.