Zhang S, Cai HQ, Ma Y *et al.* SmartPipe: Towards interoperability of industrial applications via computational reflection. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(1): 161–178 Jan. 2020. DOI 10.1007/s11390-020-9694-z

SmartPipe: Towards Interoperability of Industrial Applications via Computational Reflection

Su Zhang[1], Hua-Qian Cai[1], *Member*, *CCF*, Yun Ma[2], *Member*, *CCF*, *ACM*, *IEEE*, Tian-Yue Fan[3] Ying Zhang[4,*], *Member*, *CCF*, *ACM*, *IEEE*, and Gang Huang[1,*], *Member*, *CCF*, *ACM*, *IEEE*

[1] *Key Laboratory of High-Confidence Software Technology (Peking University), Ministry of Education Beijing 100871, China*

[2] *School of Software, Tsinghua University, Beijing 100084, China*

[3] *Hengyi Petrochemicals CO., LTD., Hangzhou 311215, China*

[4] *National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China*

E-mail: {samsuzhang, caihq}@pku.edu.cn; yunma@tsinghua.edu.cn; fty@hengyi.com {zhang.ying, hg}@pku.edu.cn

Received May 7, 2019; revised September 15, 2019.

Abstract With the advancement of new information technologies, a revolution is being taken place to bring the industry into a new era of intelligent manufacturing. One of the key requirements of intelligent manufacturing is the interoperability of industrial applications. However, it is challenging to realize the interoperability for legacy industrial applications due to 1) the deficient semantic information of data transmitted over heterogeneous communication protocols, 2) the difficulty to understand the complex process of business logic with no source code, and 3) the high cost and potential risk of reengineering the applications. To address the issues, in this paper, we propose an approach named SmartPipe to exposing existing functionalities of an industrial application as APIs without source code while simultaneously allowing the application to remain unchanged. We design a behavioral runtime model (BRM) as the self-representation of the industrial applications, based on which a computational reflection framework is designed to flexibly construct the model and generate APIs that encapsulate specific functionalities. We validate SmartPipe on a real industrial application that controls the spin-draw winding machine. Results show that our approach is effective and more suitable for industrial scenes compared with traditional approaches.

Keywords computational reflection, runtime model, interoperability, industrial application, API generation

1 Introduction

Information technologies have driven the way of manufacturing from electronic-based to information-based where industrial application software is developed for and deployed in the industrial field to improve the level of development planning, transaction processing, production scheduling and process control.

With the advancement of new information technologies such as Cloud Computing, Big Data, Artificial Intelligence, and Internet of Things, a revolution is being taken place to bring the industry into a new era of intelligent manufacturing where a highly flexible intelligent manufacturing environment is built in order to respond rapidly to changes in demand at low cost to the firm without damage to the environment [1]. Many countries have proposed national strategies to acceler-

Regular Paper

Special Section on Applications

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004800, the National Natural Science Foundation of China under Grant No. 61725201, Beijing Municipal Science and Technology Project under Grant No. Z171100005117002, and Tianjin Municipal People's Government Port Service Office (Project on Cross-Border E-Commerce Big Data Analysis and Display System).

*Corresponding Author

ate the intelligent manufacturing. For example, the United States proposed Industrial Internet [2] in 2012, and Germany proposed Industry 4.0 [3] in 2013.

One of the key requirements of intelligent manufacturing is the interoperability of industrial applications. In order to build a flexible manufacturing environment, all information about the manufacturing process should be available when it is needed, where it is needed, and in the form that it is needed across entire manufacturing supply chains, complete product lifecycles, multiple industries, and small, medium and large enterprises. Therefore, as the medium of controlling the manufacturing process in the cyber space, industrial applications should be interoperable to enable the opening, sharing and exchanging of manufacturing information.

Generally, a well-recognized way of realizing software interoperability is to expose existing functionalities as application programming interfaces (APIs) by refactoring the application implementation. However, the main obstacle to achieve the interoperability of industrial applications is that the support of the original application developers is difficult to be obtained or even lost. As a result, generating APIs for such so-called legacy industrial applications faces the following three challenges.

First, legacy industrial applications usually have no source code. As a result, it is difficult to understand the complex process of business logic of functionalities.

Second, industrial applications are usually directly connected to physical devices and the data is transmitted over heterogeneous communication protocols. As a result, the semantics of industrial applications is deficient.

Third, industrial applications are bug-sensitive where the production line failure caused by bugs or the termination of applications can be very costly. As a result, reengineering of industrial applications should take serious.

To address these challenges, our idea is to leverage computational reflection to construct a runtime model that could represent the behaviors of industrial applications. Computational reflection [4] is the ability of a computational system that provides an accurate representation of itself (called self-representation), which requires that the states and behaviors of the system are always compliant with the representation and changes made on the representation will be immediately mirrored to changes of the actual states and behaviors of the system. Based on computational reflection, we can control the behavior of an application by manipulating the high-level model without source-code level reengineering. The manipulation of the high-level model shields the complexity of the application. The control of the application maintains the high-level semantics attached by the data processing of the application. Therefore, it is promising to expose existing functionalities of legacy industrial applications as APIs without source code via computational reflection.

To this end, in this paper, we propose an approach named SmartPipe to generating APIs for industrial applications based on computational reflection. In order to describe the execution behavior of an application and the data required for the execution, we define and formalize a behavioral runtime model (BRM) that consists of an execution variation model and a data variation model as the self-representation of the application behaviors. Based on BRM, we design and implement a computational reflection framework to support the ability to reify the application behavior to its self-representation, manipulate the self-representation, and reflect the manipulations in the behavior of the application. With the framework, developers can construct the BRM of the runtime behavior of an application, generate model fragments corresponding to specific functionalities of the application, and transform the model fragments to APIs that encapsulate target functionalities according to interoperability requirements without source-code level reengineering.

The contributions of this paper are listed as follows.

1) We propose an approach named SmartPipe to exposing existing functionalities of an industrial application as APIs without source code via computational reflection while simultaneously allowing the application to remain unchanged, which can greatly improve the interoperability of industrial applications.

2) We define and formalize a behavioral runtime model (BRM) as the self-representation of an application.

3) We design and implement a computational reflection framework to flexibly construct the BRM of an application and generate APIs that encapsulate specific functionalities without source-code level reengineering.

4) We validate SmartPipe on a real industrial application that controls the spin-draw winding machine and carry out some evaluations. Results show that our approach is effective and more suitable for industrial scenes compared with traditional approaches.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 overviews SmartPipe. Section 4 introduces the computational reflection

framework based on BRM. Section 5 presents the API generation via computational reflection. Section 6 validates SmartPipe on a real industrial application and carries on some evaluations. Section 7 discusses the limitations of SmartPipe. Finally, Section 8 summarizes the paper and prospects for the future research directions.

2 Related Work

To the best of our knowledge, this paper makes the first attempt to generate APIs that expose existing functionalities of legacy industrial applications via computational reflection. We review the existing work related to the integration of legacy industrial systems, API generation, and computational reflection.

2.1 Integrating Legacy Industrial Systems

Feldhorst *et al.*[5] suggested a solution for the integration of legacy industrial systems, which distinguishes three abstract system layers: device layer, integration layer and control layer. Givehchi *et al.*[6] proposed an interoperability layer requiring no changes on the legacy device that maps field device data into an ISA95-based information model to migrate legacy industrial systems in a cost-effective manner to the new paradigm of integrated IT-OT levels. Tao *et al.*[7] proposed the Industrial Internet-of-Things Hub to realize smart interconnection in dealing with heterogeneous equipment, quick configuration and implementation, and online service generation, in which a set of flexible CA-Modules compatible with different communication interfaces and protocols are designed. These approaches depend on manual adaptation to heterogeneous interfaces and protocols of legacy industrial systems, and are more architectural than technical. What is more, the direct connection with the communication protocol can only obtain the underlying raw data, which means that the semantic information attached in the processing process of the upper control application is lost.

2.2 Generating APIs During the Development Phase

Queirós[8] presented an automatic generator of RESTFul Web applications named Kaang that generates the main API content based on the user's input and a set of templates, which will help developers to manage and test routes, define resources, store data models and others. Ed-Douibi *et al.*[9] presented an approach that leverages on MDE techniques to generate RESTful services, which takes Eclipse Modeling Framework (EMF) data models as input and generates RESTFul Web APIs relying on well-known libraries and standards, thus facilitating its comprehension and maintainability. Zhai *et al.*[10] developed a novel technique that can construct models for Java API functions by analyzing the documentation. These approaches are not applicable to generate APIs from legacy systems.

2.3 Generating APIs for Legacy Systems

Related approaches to generating APIs for legacy systems can be divided into two categories: 1) wrapping approaches that concentrate on the interface of the legacy system and hide the complexity of its internals; 2) reengineering approaches that analyze and adjust an application in order to represent it in a new form[11].

2.3.1 *Wrapping Approaches*

Stroulia *et al.*[12,13] and Canfora *et al.*[14,15] proposed wrapping approaches based on understanding and modeling the users' interaction with the legacy application interface. These approaches are highly coupled to the user interface, disturbing the operation of technicians to a great extent when the external system interacts with the legacy system.

Rodríguez-Echeverría *et al.*[16] presented a model-driven approach for deriving a REST API from a legacy web application within the frame defined by a modernization process. Jiang and Stroulia[17] proposed an approach that exploits a reverse engineering technique for modelling the interaction of the user with Web sites and obtaining the functionalities to be specified in terms of WSDL specifications to construct Web services. Baumgartner *et al.*[18] proposed a suite for obtaining Web services from web applications. These approaches are all oriented to web applications while most of industrial applications are not web applications.

Sneed[19,20] discussed a tool-supported approach to identifying and exposing individual business functions in the programs as web services by clustering and data flow analysis based on the legacy code. Lewis *et al.*[21,22] and Smith[23] discussed a migration technique that helps organizations evaluate the potential for converting components of an existing system into services. Inaganti and Behara[24] proposed an approach to migrating existing applications, which establishes the business process model, identifying the points of

functionality and exposing them as services. These approaches typically work with a small subset of the functionalities of legacy systems due to the inappropriate application architecture and most of them are based on the analysis of the source code.

Del *et al.*[25] proposed an approach to identifying pieces of functionality to be potentially exported as services from database-oriented applications by clustering queries dynamically extracted by observing interactions between the application and the database through formal concept analysis. Yeh *et al.*[26] presented a process that extracts an extended entity-relationship diagram from a table-based database with few descriptions for the fields in its tables and no description for keys. Strobl *et al.*[27] described the industrial experience in performing database reverse engineering on a large-scale application reengineering project. These approaches are oriented to database (DB)-based systems while industrial control applications access data in a different way.

2.3.2 *Reengineering Approaches*

Zhang *et al.*[28, 29] proposed a reengineering approach which applies an improved agglomerative hierarchical clustering algorithm to restructure legacy code and to facilitate legacy code extraction for Web service construction. Chen *et al.*[30] proposed a reengineering approach by identifying system features, constructing a feature model to organize the identified features, and identifying their implementation in the legacy system through feature location techniques. Guo *et al.*[31] proposed a reverse engineering technique to make the functionalities of a client-server .NET application available as Web services. Shimin *et al.*[32] presented a componentization framework for extracting reusable components from a Java system in a low cost but high precision way based on the automated class dominance analysis and domain knowledge. Cuadrado *et al.*[33] proposed a process for recovering legacy system architecture based on modifying the existing legacy code. Marchetto and Ricca[34, 35] presented a stepwise approach based on testing that can help a developer to migrate an existing Java system into an equivalent service-oriented system. Many of these approaches require source code, which is typically unavailable. In addition, in consideration of that the production line failure can be very costly in the industrial production environment, industrial enterprises tend to avoid reengineering the legacy systems.

2.4 Computational Reflection

Huang *et al.*[36, 37] presented a reflection-based approach to autonomic computing middleware, which shows the philosophy that autonomic computing should focus on how to reason while reflective computing supports how to monitor and control. The Runtime Application Architecture (RSA) based on reflective middleware is proposed to support architecture-based application maintenance and evolution. Albertini *et al.*[38, 39] proposed an approach to exploring and interacting with SystemC models by means of an introspection technique known as computational reflection. Lopez *et al.*[40] presented a blackbox solution to convert legacy single-user applications to collaborative multi-user tools by intercepting user interface libraries and input events. Bellman *et al.*[41] proposed an approach that combines the use of active experimentation driven by internal processes in the system itself and computational reflection for developing trustworthy and adaptable complex systems. These researches inspired us that existing functionalities of a legacy industrial application can be exposed without the source code while simultaneously allowing the application to remain unchanged via computational reflection.

3 Approach Overview

Simply put, the computational reflection of an application is the mapping of the running state of an application to a set of operational data. The former part constitutes the base-level entity, and the latter part constitutes the meta-level entity, while the two-way causal association is maintained between the base-level entity and the meta-level entity. Fig.1 shows the concept of computational reflection. Based on computational reflection, we can control the behavior of an application by manipulating the high-level model.

We propose an approach named SmartPipe to exposing existing functionalities as APIs without source code while simultaneously allowing the application to remain unchanged via the computational reflection framework based on the BRM.

In order to describe the execution behavior of an application and the data required for the execution, we design BRM that consists of an execution variation model and a data variation model as the self-representation of the application behaviors.

The computational reflection framework is a framework that supports the ability to reify the application

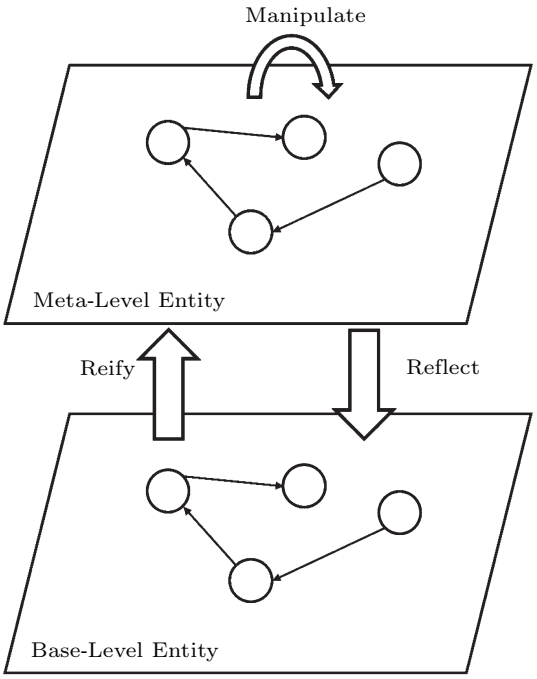


Fig.1. Concept of computational reflection.

behavior to its self-representation, manipulate the self-representation, and reflect the manipulations to the self-representation in the behavior of the application.

With the computational reflection framework, developers can construct the BRM of the runtime behavior of an application, generate model fragments corresponding to specific functionalities of the application, and transform the model fragments to APIs that encapsulate target functionalities according to interoperability requirements without reengineering.

The main steps of SmartPipe are shown in Fig.2.

1) The developer operates the target functionalities of the application through user interface in the computational reflection runtime.

2) The computational reflection runtime monitors the execution of the application and constructs the BRM.

3) The developer manipulates the BRM according to the interoperability requirements and generate APIs that encapsulate the target functionalities from alternative model fragments.

4 Computational Reflection Framework Based on Behavioral Runtime Model

Computational reflection [4] is the ability of a computational system that provides an accurate representation of itself (called self-representation), which requires that the states and behaviors of the system are always compliant with the representation (called causal-connection) and changes made on the representation will be immediately mirrored to the changes of the actual states and behaviors of the system.

Simply put, the computational reflection of an application is the mapping of the running state of an application to a set of operational data. The former part constitutes the base-level entity, and the latter part constitutes the meta-level entity, while the two-way causal association is maintained between the base-level entity and the meta-level entity.

In general, the computational reflection can be divided into structural reflection and behavioral reflection depending on the base-level entity [42]. Typically, the implementation of behavioral reflection is more difficult than structural reflection.

• The base-level entity of the structural reflection is the current program and its abstract data types, which can be regarded as the state of the application.

• The base-level entity of the behavioral reflection is the execution behavior of the current program and the data required for its execution, which can be regarded as the behavior of the application.

In order to describe the behavior of applications and the interaction between modules, behavioral reflection is necessary for complex applications. Therefore, we design a computational reflection framework based on behavioral runtime model (BRM) to support the behavioral reflection, which is shown in Fig.3.

In the proposed framework, the BRM is a self-representation of the behavior of an application, reifying it in a complete, accurate and flexible manner, which can be constructed automatically with some configuration. Developers can easily manipulate the self-representation based on algorithmic aids to generate model fragments corresponding to specific functionalities of the application. After that, the manipulations can be reflected in the behavior of the application. That is, the model fragments can be transformed to APIs that encapsulate target functionalities according to interoperability requirements semi-automatically, achieving the control of the application.

4.1 Behavioral Runtime Model

When an application is to run in an operating system, the operating system loads the required executable files into memory and starts executing. In general, the memory occupied by a process can be divided into four areas.

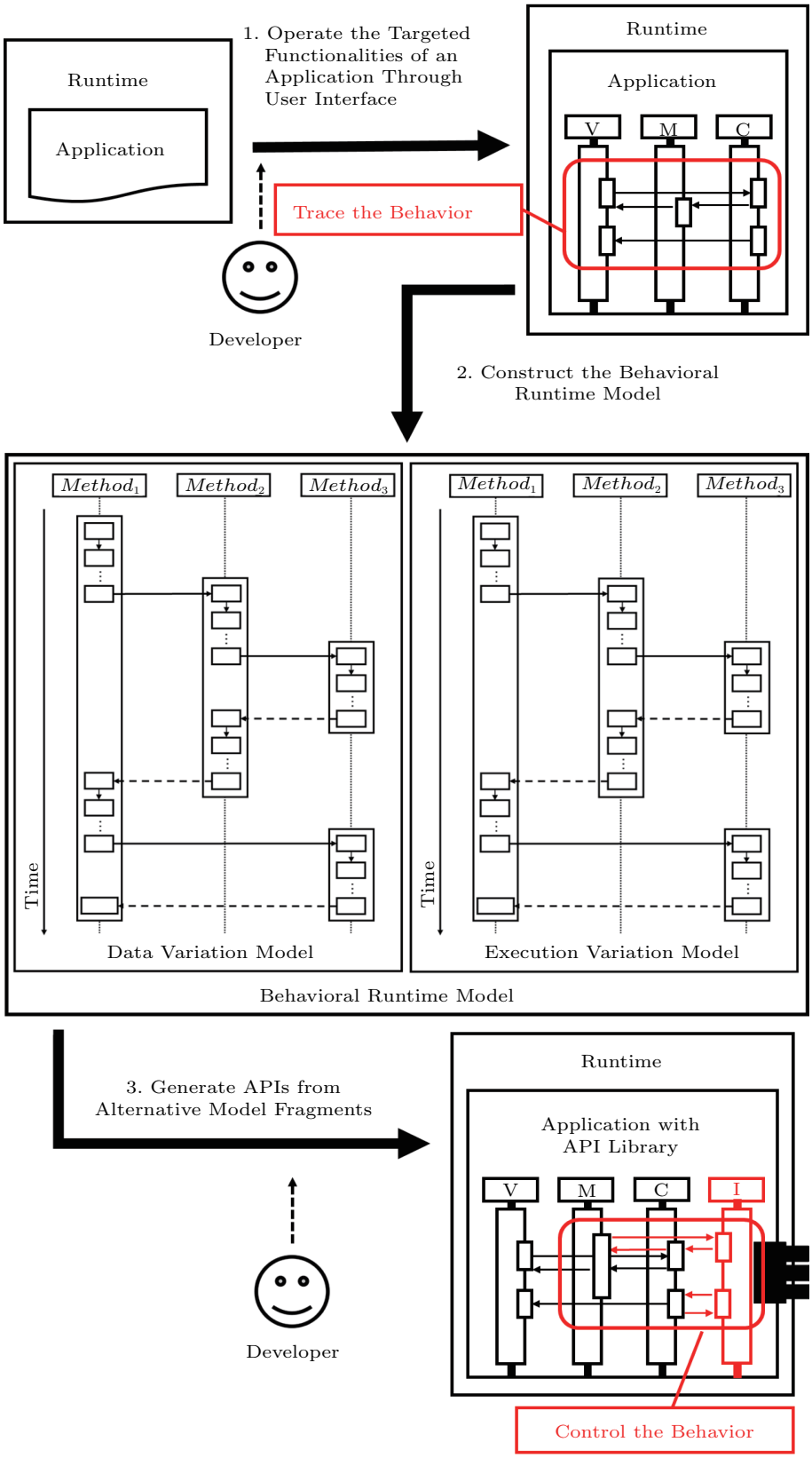


Fig.2. Overview of SmartPipe.

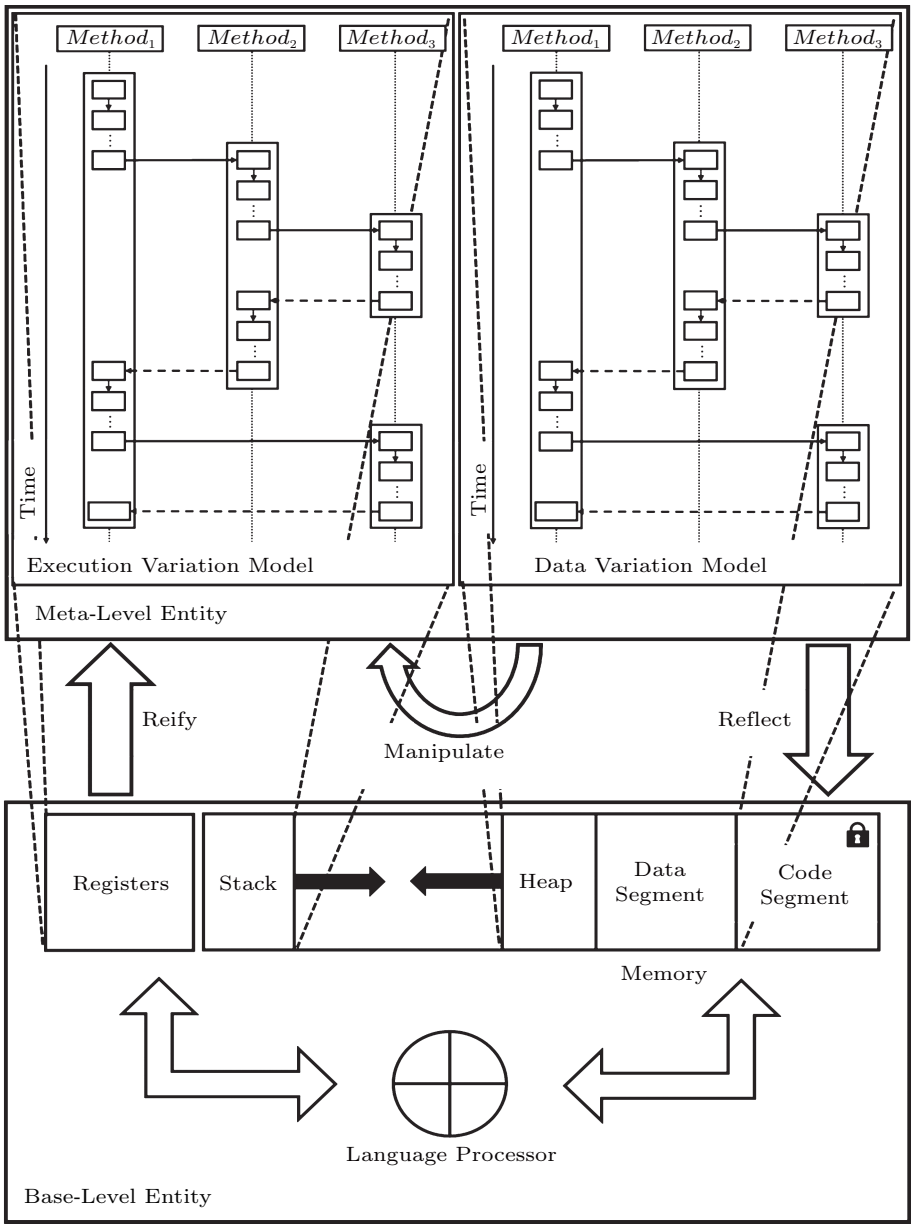


Fig.3. Computational reflection framework based on BRM.

• *Code Segment.* A code segment, also known as a text segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

• *Data Segment.* A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

• *Heap.* The heap is the segment where dynamic memory allocation usually takes place.

• *Stack.* The stack area contains the program stack, a last-in-and-first-out structure, typically located in the higher parts of memory. The stack area traditionally adjoined the heap area and grew in the opposite direction.

When an application is running, the execution of code segment will cause changes in other memory area and registers. The BRM of an application should be able to react to the application over a period of time in two aspects: the execution behavior of the application and the data required for its execution.

We formally define the BRM in following subsections.

4.1.1 Formal Definition

The behavioral runtime model (BRM) is a self-representation of the application behavior that consists of the code execution and the data changes.

Definition 1. *The behavioral runtime model consists of an instruction set, a set of local data state, a set of global data state, a relationship that denotes the order or dependence of two certain executed instructions and the local data state between the execution of them, and a relationship that denotes the change from one global data state to the other and the instruction(s) that caused the change.*

$$M = (I, L, G, \varepsilon, \sigma),$$

where

- $I = \{i_i\}$ *is the instruction set, where* i_i *is an executed instruction consisting of an opcode and some operands;*
- $L = \{R_i, S_i\}$ *is the set of local data state, where* $R_i = \{(register, data)_{ij}\}$ *is the set that denotes the register state and* $S_i = \{(address, data)_{ij}\}$ *is the set that denotes the memory state of stack area;*
- $G = \{D_i, H_i\}$ *is the set of global data state, where* $D_i = \{(address, data)_{ij}\}$ *is the set that denotes the memory state of data segment and* $H_i = \{(address, data)_{ij}\}$ *is the set that denotes the memory state of reachable heap area;*
- $\varepsilon \subseteq I \times I \times L$ *denotes the order or dependence of two certain executed instructions and the local data state between the execution of them;*
- $\sigma \subseteq G \times G \times I$ *denotes the change from one global data state to another and the instruction(s) that caused the change.*

According to the base-level entity of behavioral reflection, the BRM can be divided into two parts: the execution variation model that describes the execution behavior of the application and the data variation model that describes the data required for the execution.

What is more, considering that the behavior of a functionality of the application can be comprehended better through the sequences of method call, and the accuracy and completeness of the behavior can be guaranteed through the sequences of instruction execution, we divided the execution variation model and the data variation model into two levels: the instruction level and the method level.

4.1.2 Execution Variation Model

The execution variation model is used to describe the execution behavior of the application during a period of time, which is shown in Fig.4. With the execution variation model, developers can inspect the running state of the instructions at any time.

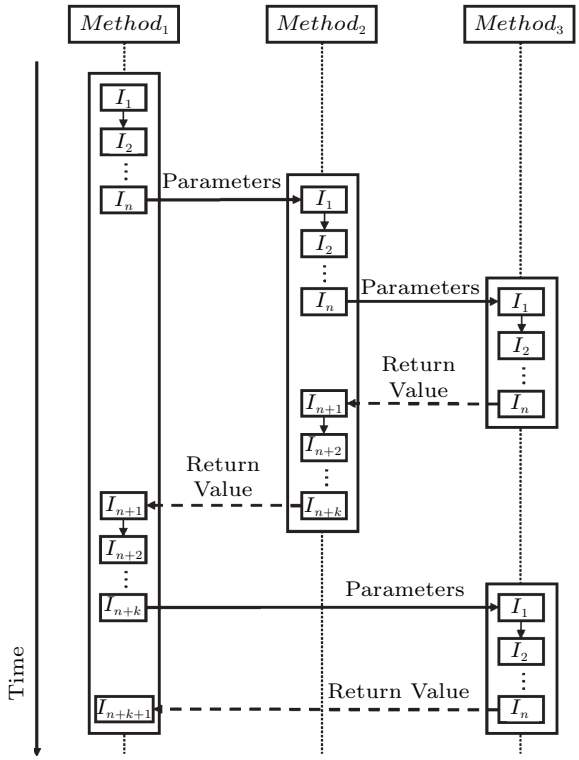


Fig.4. Execution variation model.

Definition 2. *The execution variation model* $M_{\text{execution}}$ *consists of an instruction set, a set of local data state, and a relationship that denotes the order or dependence of two certain executed instructions and the local data state between the execution of them.*

$$M_{\text{execution}} = (I, L, \varepsilon).$$

In an instruction-level execution variation model, the instructions in I are all executed instructions. In a method-level execution variation model, the instructions in I are all method call and method return instructions. Obviously, $I_{\text{method-level}} \subset I_{\text{instruction-level}}$.

4.1.3 Data Variation Model

The data variation model is used to describe the data required for the execution during a period of time, which is shown in Fig.5. With the data variation model, developers can inspect the state of accessible variables in data segment and heap area at any time.

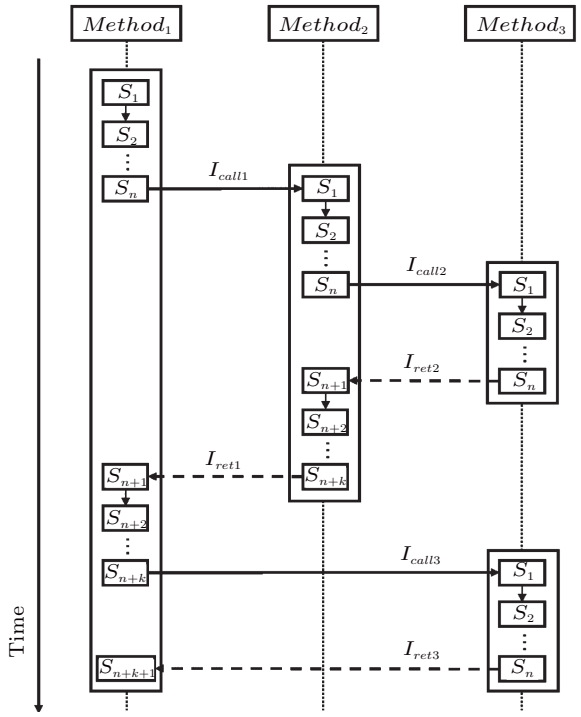


Fig.5. Data variation model.

Definition 3. *The data variation model* M_{data} *consists of an instruction set, a set of global data state, and a relationship that denotes the change from one global data state to another and the instruction*(*s*) *that caused the change.*

$$M_{\text{data}} = (I, G, \sigma).$$

In an instruction-level data variation model, the instruction in σ is one certain effective instruction (an instruction that has effect on accessible variables in data segment and heap area). In a method-level data variation model, the instructions in σ are some certain effective instructions between two method instructions (method call or method return), that is, each element in G denotes the global data state when a certain method instruction was executed. Obviously, $G_{\text{method-level}} \subset G_{\text{instruction-level}}$.

4.2 Constructing the Behavioral Runtime Model

The construction of the BRM includes the following challenges.

1) The execution of an application is complex. How to achieve the fine-grained instructions that are able to represent the execution of an application?

2) Multi-thread is commonly applied to provide better user experience. How to record the dependent relationship between different control flows?

3) According to the definition, the BRM covers every variation of the execution behavior and the required data of an application over a period of time. How to solve the problem of state explosion when modeling?

To overcome the above challenges, we take the following measures.

1) The behavior interpreter is an interpreter that interprets executable instruction, which can monitor the instructions at runtime and simulate the execution of recorded instructions. By implementing the behavior interpreter, we can achieve the complete, accurate and detailed records of the application behavior. What is more, we hooked up some system methods that send system messages to form handles, which means that the interactions between users and the application are also included in the method-level execution variation model.

2) Typically, a control flow is represented by a thread, and the dependence can be divided into two categories: synchronization dependence and communication dependence, which can be inferred during the interpretation process and represented by ε in the execution variation model.

• Synchronization dependence captures the dependent relationship due to inter-thread synchronization. Informally, an instruction i_a in one thread is synchronization dependent on an instruction i_b if the start and(or) termination of the execution of i_a determinate(s) the starts and(or) termination of the execution of i_b through an inter-thread synchronization. Typically, synchronization dependence is associated with method call and method return instructions.

• Communication dependence represents dependent relationship due to inter-thread communication. Informally, an instruction i_a in one thread is directly communication-dependent on an instruction i_b in another thread if the struct used at i_b has direct influence on the struct used at i_a through an inter-thread communication.

3) Considering the terribly high time and space complexity, it is infeasible to snapshot the state of registers, stack, heap, and data segment whenever something changes. Therefore, we proposed a rolling forward strategy to relieve the problem of state explosion.

4) In addition to the behaviors related to the target functionality, there are also some irrelevant behaviors to be eliminated, such as the regular check to the network connection. We proposed a three-level filter to further relieve the problem of state explosion.

4.2.1 Key Mechanism 1: Rolling Forward Strategy

To avoid the terribly high time and space complexity caused by snapshotting the data state of every variation, we propose the rolling forward strategy. The basic idea of the rolling forward strategy is to restore the state of registers, stack, heap, and data segment on demand based on the initial state and the subsequent instructions.

Considering that the strategy of rolling forward from the initial state to the required state all the time may be inefficient, we trade off between the recording cost and the querying cost by adjusting the strategy to record the runtime state every time when a method is called or returns, and restore the required state from the latest recorded state.

Algorithm 1 shows the rolling forward algorithm. The behavior interpreter is used to record the required information and support the rolling forward algorithm.

Algorithm 1. Rolling Forward Algorithm

Input: the behavioral runtime model, M; the instruction, i; the target state type, t

Output: the state of t when i was to execute in M

```
Instruction i_method <- Find the information about the latest method call or method return instruction from M.Instructions.
State s <- Get the snapshot of the state of registers, stack, heap, and data segment when i_method is to be executed from M.
Instructions I <- Find the collection of instructions executed between i_method and i from M.Instructions.
for instruction i_cur in I do
    Execute i_cur on s by the behavior interpreter. Update s.
State s_ret <- Get the required state from s by t.
return s_ret
```

4.2.2 Key Mechanism 2: Three-Level Filter

The number of nodes in a fine-grained record is the number of instructions executed during this time, which greatly increases the complexity of model generation and the difficulty of model analysis. On the one hand, the complexity of runtime instructions will be tremendously increased if each execution of an instruction is treated as a runtime instruction in the BRM. On the other hand, the accuracy of the model will be compromised if we ignore indispensable parts of the execution. Therefore, we implement a three-level filter to control the complexity of the model and meet the granularity requirement in different scenarios, including an interaction-level filter, a method-level filter, and an instruction-level filter.

The interaction-level filter filters behaviors irrelevant to a specific functionality by the type and title of controls (e.g., button, textbox). As mentioned before, the system messages that represent the interaction between users and the application are included in the method-level execution variation model. When a user interacts with an application to trigger a specific functionality, the events of mouse and keyboard usage will be converted to system messages to some specific controls of the application. These messages then trigger the execution of some specific response methods of these controls, which begin the execution of the functionality. Therefore, the scale of the BRM can be narrowed down to a specific functionality by analyzing the correlation between the action of users and the reaction of application.

Besides, the method-level filter filters the methods unconcerned by developers by method name and the instruction-level filter filters the unconcerned instruction by instruction type. Developers can configure the three-level filters on demand to generate a BRM of appropriate granularity in different scenarios.

5 API Generation via Computational Reflection

The BRM is a self-representation of the behavior of an application, which can comprehensively and accurately describe the runtime behavior of an application. Computational reflection requires that the self-representation is manipulable, which means the manipulations to the self-representation should be reflected in the behavior of the application.

In SmartPipe, developers manipulate the BRM by generating model fragments corresponding to specific functionalities, and the manipulations can be reflected in the behavior of the application by transforming the model fragments to APIs that encapsulate target functionalities according to interoperability requirements semi-automatically, thus achieving the control of the application.

The generation of APIs includes following challenges.

1) A complete functional execution process is usually highly coupled to the user interface, thus an model fragment that corresponds to the target functionality while is low-coupled to the user interface should be located.

2) Due to the compilation and the possible confusion, the method names are usually confused and have

no semantic information, which greatly increases the difficulty of locating the appropriate model fragment.

3) To ensure that the effects of the generated API are consistent with the target functionality, the context of the model fragment should be well constructed.

To overcome the above challenges, we take the following measures.

1) We propose a keyword-based contamination algorithm to solve the problems of semantic missing and user-interface coupling during the location of appropriate model fragments.

2) We propose an approach to restoring the necessary data dependence of the model fragment while ensuring the flexibility of APIs.

5.1 Generation of Model Fragments

In response to the problems of semantic missing and user-interface coupling, we propose a keyword-based contamination algorithm. Although the semantic information is usually lacking in the BRM, the values of runtime data instances are unambiguous. The core idea of the keyword-based contamination algorithm is to use the information related to the functionality execution provided by developers to sort the structs in memory by their correlation with the provided information and sort the alternative model fragments by their correlation with the sorted structs and the provided information. The algorithm is shown in Algorithm 2.

The first step of the algorithm is to build a struct diagram (line 1). After that, the algorithm iterates through all the instructions and reference chains in the BRM to update the weight of the struct diagram (lines 2–4), where the value of α is 0.8 and $Count1(i,j)$ represents the number of instructions that include both node i and node j.

After constructing the struct relationship diagram, the algorithm proceeds to the second step: performing keyword-based matching to find structs that are directly related to the keyword (lines 5–7), where $Count2(i,k)$ represents the number of instructions that contain both node i and keyword k, and β is 2 000. In this step, the keyword is a string or value provided by the developer, which is the special data in the process of executing a certain function. For example, the corresponding special data of a BRM describing the process of querying production data can be the content of the query results.

Algorithm 2. Keyword-Based Contamination Algorithm

Input: the behavioral runtime model, M; the keyword, k
Output: the alternative model fragments sorted by score

1 Build the struct diagram $< V, E >$ whose nodes and edges are weighted with $W_{\mathrm{v}}(i) \leftarrow 0$ and $W_{\mathrm{e}}(i,j) \leftarrow 0$, $i,j \in V$ from M_h.
2 **for** instruction *inst* in $M.instructions$ **do**
3 **if** instruction *inst* contains structs i,j **then**
4 $W_{\mathrm{e}}(i,j)$ and $W_{\mathrm{e}}(j,i) \leftarrow 1-\alpha^{Count1(i,j)}$
5 **for** instruction *inst* in $M.instructions$ **do**
6 **if** instruction *inst* contains struct i and keyword k **then**
7 $W_{\mathrm{v}}(i) \leftarrow \beta \times Count2(i,k)$
8 **for** $p \leftarrow 1$ to n **do**
9 **forall** structs i,j such that $W_{\mathrm{e}}(j,i) > 0$ **do**
10 $W_{\mathrm{v}}(j) \leftarrow W_{\mathrm{v}}(j) + W_{\mathrm{v}}(i) \times N(p) \times W_{\mathrm{e}}(i,j) \times \frac{Avg(i)}{Total(i)}$
11 $W_{\mathrm{v}}(i) \leftarrow W_{\mathrm{v}}(i) \times (1 - N(p) \times Avg\,(i))$
12 **for** instruction *inst* in $M.instructions$ **do**
13 **if** instruction *inst* is a read(write) instruction for a field in memory **then**
14 $inst.correlation \leftarrow \sum_i V(inst.struct[i])$
15 **if** instruction *inst* is releted to user interface **then**
16 $inst.correlation \leftarrow inst.correlation - Penalty$
17 **for** model fragment mf in $M.modelFragments$ **do**
18 $mf.score \leftarrow$ $\sum_i W_{\mathrm{v}}(mf.struct[i]) + \sum_i (mf.instruction[i]).correlation$
19 Sort the model fragments $M.modelFragments$ by *score*.
20 **return** $M.modelFragments$

After initializing the weight of the directly related struct, the algorithm proceeds to the third step: contaminating the indirectly related structs (lines 8–11), where $N(p) = 0.5p + 1$ is used to control the total amount of the weight change of different nodes in each iteration and $E(i,j) \times \frac{Avg(i)}{Total(i)}$ is the amount of contamination of each successor node according to the proportion of the weight of the edge.

After calculating the correlation of structs, we can calculate the correlation of the instructions in the execution variation model and convert the model to reduce the execution logic associated with the user interface (lines 12–16), where the penalty parameter $Penalty = 0.5$.

After calculating the correlation of each instruction, the correlation of the specific model segment with the specified structs in each control flow can be calculated and a number of alternative model fragments sorted by the correlation can be generated (lines 17–19). Due to the presence of the penalty coefficient, the correlation of the model fragments associated with the user interface is typically less than that of other ones.

Deletion is the main operations in this process, which may be accidentally wrong because of the possi-

ble underreporting of the keyword-based related struct contamination analysis. Therefore, the scope of the contamination can be expanded by adjusting $N(p)$ and the number of iteration rounds in the analysis algorithm when developers find that there is no suitable model fragment among the recommended model fragments.

5.2 Construction of Data Dependence

A given fragment of BRM consists of a set of instructions. Each model fragment can automatically generate a code snippet that maintains the control equivalence of heap to the original fragment through program analysis. The main challenge of generating the code snippet is the construction of the data dependence of the model fragment, which should keep the data that the model fragment depends on the same as when it executed.

The types of data dependence of a model fragment can be divided into two types: the numeric type and the struct type.

1) For the data dependence of numeric type, since the data type in the BRM is a concretized value, the dependence of the numeric type in the newly generated code snippet is consistent with the value in the BRM.

2) Data dependence of struct type can be further subdivided into local struct type and global struct type. The local struct can be regarded as a struct that can be recycled after the execution of the BRM, and the global struct is a struct that is not affected by the execution of the function. For a local struct, the code snippet to construct it can be automatically generated by playing back the related instructions that change the struct. For a global struct, a reachable struct reference chain for obtaining the struct can be constructed by program analysis, and the code snippet to construct it can be automatically generated according to the reference type between the structs described in the chain.

5.3 Parameterization of Interactive Variables

The code snippets (i.e., APIs) generated by the model fragments are based on the monitored behaviors of an application. After the construction of data dependence, the value of variables in a code snippet is the same as that when the target functionality was executed, which means that the execution result of this code snippet is the same as the target functionality.

Interactive variables are the variables that represent the alterable input of users when they trigger the targeted functionality. For example, in the case of a production data query, the date parameter is an interactive variable, which is a fixed string in the automatically generated API. Obviously, such a kind of API is inflexible. Therefore, the interactive variables in generated APIs should be replaced with modifiable parameters. In addition, the conversion from the directly achieved data of various structures to standard data formats like JSON is necessary.

6 Case Study

In this section, we validate SmartPipe on a real industrial application that controls the spin-draw winding machine and compare it with other approaches. Our cooperative enterprise is Zhejiang Hengyi Petrochemicals Co., Ltd., China[①].

6.1 Background

In industrial fields, large industrial devices are usually equipped with dedicated control applications, with which technicians at the production site can control the behavior of the device, monitor the status of the device and view production records of the device.

For various reasons, most of these control applications do not provide a remotely callable API that encapsulates the above functions. Manufacturers using these devices can only arrange technicians to go to the operation consoles of each production floor to perform on-site operations on these control applications, which is inefficient and greatly hinders the development of intelligent manufacturing.

Fig.6 shows a dedicated application for controlling a spin-draw winding machine (a device for producing polyester filament yarn) facing with the above problem.

During the spinning process, this control application records various events that occur during the operation of the machine (including normal events such as winding start and abnormal events such as over temperature) and the production data of the machine (including statistical reports such as package report, production report and yarn break report).

Manufacturers want to remotely and automatically acquire these device data in a programmatic manner to support further data analysis (optimization of process parameters, prediction of equipment failures, etc.) and automation integration (shovelling wire with mechanical arm automatically, etc.).

[①] http://en.hengyi.com, Nov. 2019.

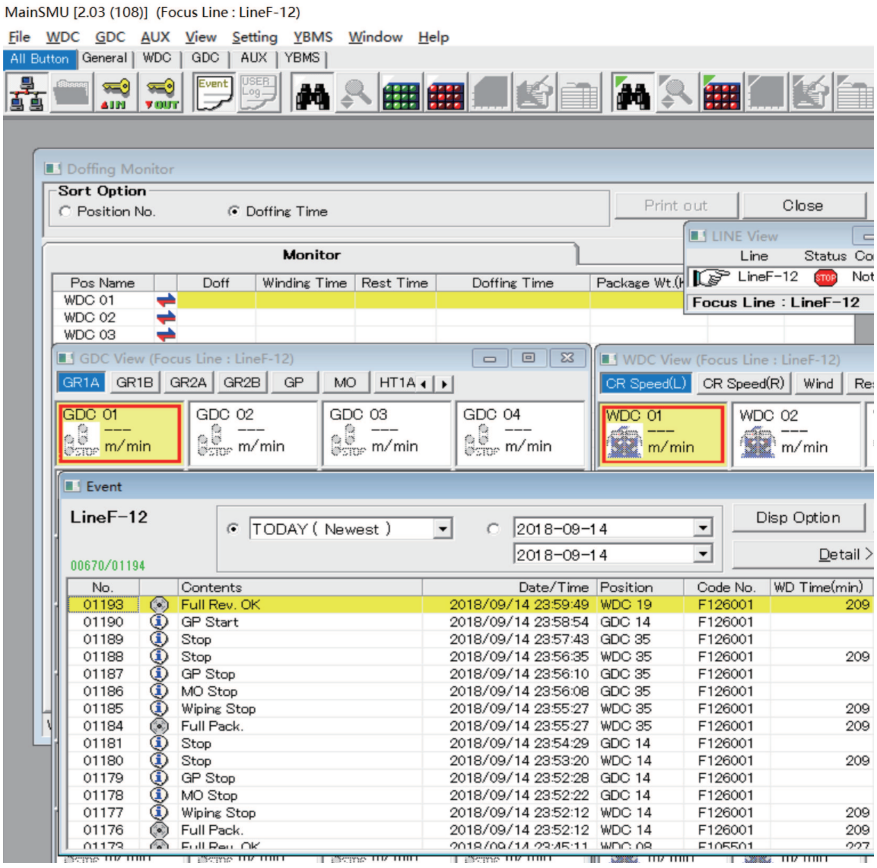


Fig.6. Program for controlling the spin-draw winding machine.

6.2 Generating Process

Through SmartPipe, we generate the corresponding APIs for the six functionalities of this control application: EventInfo, YBMSBaseInfo, PackageReport, DoffChangeReport, ProductionReport and YarnBreakReport. The following will introduce the process of developing APIs with the API for querying EventInfo as an example.

First, we configured a coarse-grained model construction strategy (only for function-level models) to monitor all modules of the application. Fig.7 shows the view of the generated model.

Based on the generated model, the contamination analysis is carried out to determine the specific modules that implement the target functionality. After that, a fine-grained model is constructed to support the generation of API. Fig.8 shows a part of the API code.

At last, the converting from date strings in the request (the start date and the end date of a query request) to the integers in the method (*param*1 and *param*2) and the converting from the result struct (*res*3) to the JSON string in the response should be implemented by developers with the help of the BRM. The execution result of the developed API is shown in Fig.9.

We developed the remaining five APIs in similar ways. Thus, the device data enclosed in the control application is opened through APIs via computational reflection, which greatly improves the interoperability of the application.

6.3 Comparison with Other Approaches

The application mentioned above is a legacy desktop application without database. The source code is unavailable, the communication protocol is unknown, and the outdated architecture is not suitable for being directly exposed as APIs. Therefore, most of the approaches mentioned in Section 2 cannot be implemented except wrapping approaches based on user interface.

We implemented two APIs that encapsulate the same functionality of querying the package report: one based on computational reflection (CR-based API) and the other based on user interface (UI-based API), and compared their execution efficiency, concurrency,

development time and impact on application operation. Table 1 shows the results of the comparison.

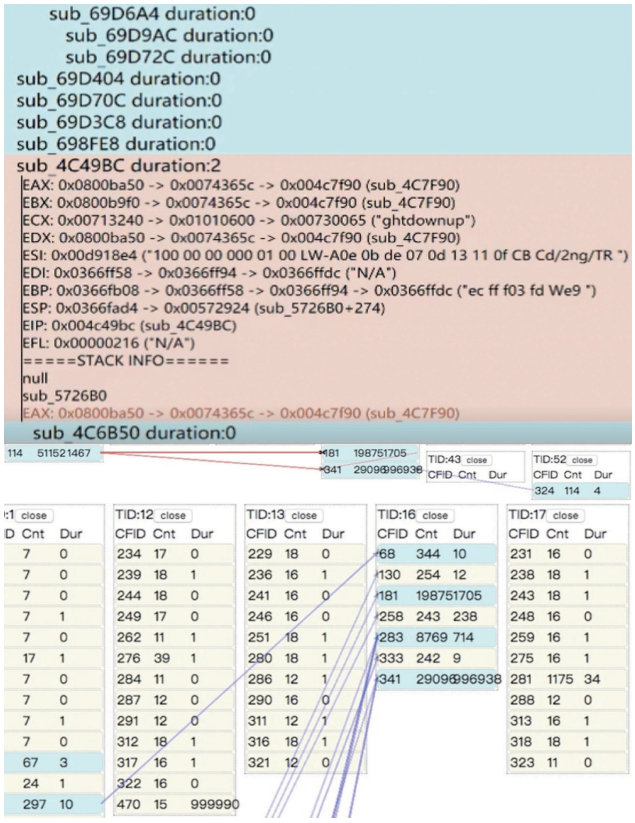


Fig.7. View of method-level execution variation model.

```
5   typedef void*(__fastcall *callmehod1)(int, int);
6   typedef void*(__fastcall *callmehod2)(char*);
7   typedef void*(__fastcall *callmehod3)(int, int, char*);
8
9   static void* queryEventInfo(int param1, int param2){
10      int addr1 = 0x60F9D0;
11      //myLogger("\nStart function at 0x%x\n", addr1);
12      callmehod1 m1 = (callmehod1)addr1;
13      void* res1 = m1(param1, param2);
14      //myLogger("##Call:\n\tparam1:%d param2:%d result:%p
15      int addr2 = 0x67300C;
16      //myLogger("\nStart function at 0x%x\n", addr2);
17      callmehod2 m2 = (callmehod2)addr2;
18      void* res2 = m2((char*)res1);
19      //myLogger("##Call:\n\tparam1:%p result:%p\n", res1,
20      int addr3 = 0x640194;
21      //myLogger("\nStart function at 0x%x\n", addr3);
22      callmehod3 m3 = (callmehod3)addr3;
23      void* res3 = m3(param1, param2, (char*)res2);
24      //myLogger("##Call:\n\tparam1:%d param2:%d param3:%p
```

Fig.8. Part of generated code of the API.

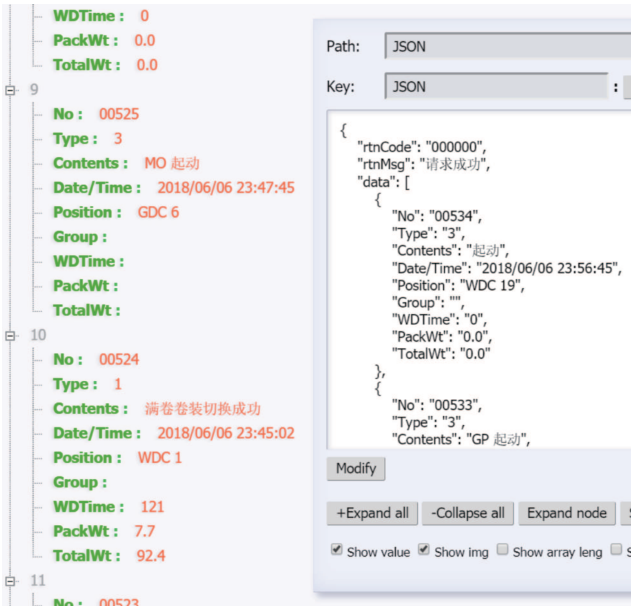


Fig.9. Execution result of the developed API.

In terms of the execution efficiency, the CR-based API is more efficient than the UI-based API. We tested the cumulative execution time of the two APIs for 1 000 single-threaded executions. The execution time does not include the communication time. The results show that the average time for the CR-based API to execute is 21.14 ms, and the average time for the UI-based API to execute is 383.48 ms, which is about 18 times that of the CR-based API. The reason is that the implementation of the CR-based API only includes the indispensable key business logic that has been stripped out from the application while the implementation of the UI-based API includes redundant processes such as the response of user interface, which is time-consuming.

In terms of the concurrence, the CR-based API supports concurrency while the UI-based API does not support concurrency due to the high coupling with the user interface. Techniques like Docker may help the UI-based API support concurrency. However, since industrial control applications are often connected to the target physical manufacturing devices, these techniques cannot be applied. We tested the average execution time of the CR-based API at 10/20/50/100

Table 1. Comparison of CR-Based and UI-Based Approaches

Implementation	Execution Efficiency (ms/request)	Concurrence	Development Time (hour) (1 developer)	Impact on Operation
CR-based	21.14	Support	8	Slight
UI-based	383.48	Not support	2	Serious

Note: CR-based means computational-reflection based. We implemented two APIs that encapsulate the same functionality of querying the package report: one based on computational reflection and the other based on user interface, and compared their execution efficiency, concurrency, development time and impact on application operation.

threads concurrency, and the results were divided into 22.90/24.43/30.63/33.58 ms, where the API was called once every 1 second and 100 times per thread. Therefore, the concurrency of the CR-based API is sufficient to support most of the scenarios, as the key business logic is stripped out appropriately.

In terms of the development time, the development of UI-based API took about 2 hours while that of the CR-based API took about 8 hours with the same developer because the UI-based API only needs to simulate the user operation and capture the changes of user interface while the CR-based API needs to strip the key business logic inside the application. However, the computational reflection framework based on the BRM provides a lot of assistance for development; thus developers can conduct the work without comprehending or modifying the internal logic of the target functionality. Therefore, the development complexity of the CR-based API is moderate.

In terms of the impact on user operation, the user cannot operate the control application normally when the UI-based API is slight called due to the high coupling with the user interface, while the impact is negligible when the CR-based API is frequently called. We tested the execution time of the UI-based API while running the above program for concurrence test (100 threads). The average time for the UI-based API to execute is 383.48 ms, which was only increased by 6.02% compared with that in the normal state.

In summary, SmartPipe is effective and practical, which is more appropriate to expose the existing functionalities of a legacy industrial application as APIs compared with traditional approaches.

7 Discussion

In this section, we discuss the limitation of SmartPipe from aspects of generality, performance and security.

7.1 Generality

We discuss the generality from programming languages, operating environment and complexity of application.

7.1.1 Programming Language of Application

Common programming languages can be divided into compiled languages (Java, C#, C, C++, etc.) and interpreted languages (Python, JavaScript, etc.).

For interpreted languages, the source code for such applications is directly available due to the nature of their interpretation. That is to say, for an application developed in an interpreted language, most of the reengineering approaches in related work can be used for API generation. In addition, industrial control applications typically do not use interpreted language development in view of efficiency issues. Therefore, we did not implement SmartPipe for an interpreted language. Of course, SmartPipe is in principle compatible with interpreted languages.

For compiled languages, it can be divided into "semi-compiled languages" (such as Java, C#, and so on) with intermediate code and "fully compiled languages" (such as C/C++) with only machine code. For "semi-compiled languages", we have achieved good support for Java and C#. For "fully compiled languages", due to the large amount of metadata loss in the compilation process, the generated model contains less type description information, which leads to the relatively high requirements for API developers.

7.1.2 Operating Environment of Application

The operating environment of industrial applications can be divided into two categories: general environment (Windows PC, etc.) and dedicated environment (PLC, etc.). SmartPipe supports the general environment in which the upper control program mainly runs, and does not support the dedicated environment.

7.1.3 Complexity of Application

SmartPipe can be applied to complex industrial applications, such as distributed industrial applications. The internal logic of the target functionality does not need to be understood or modified in SmartPipe; thus the complexity of industrial applications does not affect the usability of the method. Of course, calls to the generated APIs must follow the usage rules of the original functionalities; otherwise the execution of APIs may go wrong. In addition, the poor application architecture design, such as over-coupling between different functionalities, will increase the difficulty of development, thus reducing the generality of SmartPipe to some extent.

7.2 Performance

The performance of an API is limited by the specific implementation of the business logic of the target functionality. That is to say, the execution efficiency of an

API will be correspondingly low when that of the target functionality is low and an API does not support concurrency in cases where, for example, the key business logic involves a write operation to the same file. Since the APIs are generated by deleting the redundant execution process, the execution efficiency is theoretically not lower than that of the original functionalities and a certain extent of concurrency is supported in most cases.

7.3 Security

Although APIs generated by SmartPipe follow the original security mechanism in the application, any form of exposing functionalities as APIs poses an additional security risk: the attacker has an additional means of intrusion. Therefore, additional measures such as gateways, firewalls and encryption need to be added to ensure security when deploying APIs generated by SmartPipe.

8 Conclusions

This paper proposes SmartPipe, a computational-reflection-based approach to improving the interoperability of industrial applications by exposing existing functionalities as APIs. SmartPipe uses the behavioral runtime model as the self-representation of an industrial application and provides the computational reflection framework to flexibly construct the model and generate APIs that encapsulate specific functionalities. The main innovative feature of SmartPipe is that it is not dependent on the source code and allows the application to remain unchanged. We validated SmartPipe on a real industrial application that controls the spin-draw winding machine and carried out some evaluations to compare it with other approaches. Results showed that our approach is effective and more suitable for industrial scenes compared with traditional approaches. As future work, we plan to better support fully compiled languages and explore how to apply SmartPipe in a dedicated environment like Programmable Logic Controller.

References

[1] Shipp S S, Gupta N, Lal B *et al.* Emerging global trends in advanced manufacturing. Technical Report, Institute for Defense Analyses, 2012. https://www.nist.gov/sites/default/files/documents/2017-/05/09/IDA-STPI-report-on-Global-Emerging-Trends - in-Adv-Mfr-P-4603_Final2-1.pdf, August 2019.

[2] Lin S W, Miller B, Durand J *et al.* Industrial Internet reference architecture. Technical Report, Industrial Internet Consortium, 2015. https://www.iiconsortium.org/pdf/SHI-WAN%20LIN_IIRA-v1%208-release-20170125.pdf, Nov. 2019.

[3] Bechtold J, Lauenstein C, Kern A *et al.* Industry 4.0 — The capgemini consulting view. Technical Report, Capgemnini Consulting, 2014, 31. https://www.capgemini.com/consulting/wp-content/uploads/sites/30/2017/07/capgemini-consulting-industrie-4.0_0_0.pdf, Nov. 2019.

[4] Maes P. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 1987, 22(12): 147-155.

[5] Feldhorst S, Libert S, Ten Hompel M *et al.* Integration of a legacy automation system into a SOA for devices. In *Proc. the 12th IEEE Int. Conf. Emerging Technologies and Factory Automation*, September 2008, Article No. 110.

[6] Givehchi O, Landsdorf K, Simoens P *et al.* Interoperability for industrial cyber-physical systems: An approach for legacy systems. *IEEE Trans. Industrial Informatics*, 2017, 13(6): 3370-3378.

[7] Tao F, Cheng J F, Qi Q L. IIHub: An industrial Internet-of-Things hub toward smart manufacturing based on cyber-physical system. *IEEE Trans. Industrial Informatics*, 2018, 14(5): 2271-2280.

[8] Queirós R. Kaang: A RESTful API generator for the modern web. In *Proc. the 7th Symp. Languages, Applications and Technologies*, June 2018, Article No. 1.

[9] Ed-Douibi H, Izquierdo J L C, Gómez A *et al.* EMF-REST: Generation of RESTful APIs from models. In *Proc. the 31st Annual ACM Symp. Applied Computing*, April 2016, pp.1446-1453.

[10] Zhai J, Huang J J, Ma S Q *et al.* Automatic model generation from documentation for Java API functions. In *Proc. the 38th IEEE/ACM Int. Conf. Software Engineering*, May 2016, pp.380-391.

[11] Almonaies A A, Cordy J R, Dean T R. Legacy system evolution towards service-oriented architecture. In *Proc. the 2010 Int. Workshop on SOA Migration and Evolution*, March 2010, pp.53-62.

[12] Stroulia E, El-Ramly M, Sorenson P *et al.* Legacy systems migration in CelLEST. In *Proc. the 22nd Int. Conf. Software Engineering*, June 2000, Article No. 790.

[13] Stroulia E, El-Ramly M, Sorenson P. From legacy to web through interaction modeling. In *Proc. the 18th Int. Conf. Software Maintenance*, October 2002, pp.320-329.

[14] Canfora G, Fasolino A R, Frattolillo G *et al.* Migrating interactive legacy systems to web services. In *Proc. the 10th European Conf. Software Maintenance and Reengineering*, March 2006, pp.24-36.

[15] Canfora G, Fasolino A R, Frattolillo G *et al.* A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 2008, 81(4): 463-480.

[16] Rodríguez-Echeverría R, Macías F, Pavón V M *et al.* Model-driven generation of a REST API from a legacy web application. In *Proc. the 9th International Workshop on Model-Driven and Agile Engineering for the Web*, July 2013, pp.133-147.

[17] Jiang Y T, Stroulia E. Towards reengineering web sites to web-services providers. In *Proc. the 8th European Conf. Software Maintenance and Reengineering*, March 2004, pp.296-305.

[18] Baumgartner R, Gottlob G, Herzog M *et al.* Interactively adding web service interfaces to existing web applications. In *Proc. the 2004 Symp. Applications and the Internet*, January 2004, pp.74-80.

[19] Sneed H M, Wien A G. Wrapping legacy software for reuse in a SOA. *Multikonferenz Wirtschaftsinformatik*, 2006, 2: 345-360.

[20] Sneed H M. Integrating legacy software into a service oriented architecture. In *Proc. the 10th European Conf. Software Maintenance and Reengineering*, March 2006, pp.3-14.

[21] Lewis G, Morris E, Smith D. Analyzing the reuse potential of migrating legacy components to a service-oriented architecture. In *Proc. the 10th European Conf. Software Maintenance and Reengineering*, March 2006, pp.15-23.

[22] Lewis G, Morris E, Smith D *et al.* Service-oriented migration and reuse technique (SMART). In *Proc. the 13th IEEE Int. Workshop on Software Technology and Engineering Practice*, September 2005, pp.222-229.

[23] Smith D. Migration of legacy assets to service-oriented architecture environments. In *Proc. the 29th Int. Conf. Software Engineering*, May 2007, pp.174-175.

[24] Inaganti S, Behara G K. Service identification: BPM and SOA handshake. *BPTrends*, 2007, 3: 1-12.

[25] del Grosso C, di Penta M, de Guzman I G R. An approach for mining services in database oriented applications. In *Proc. the 11th European Conf. Software Maintenance and Reengineering*, March 2007, pp.287-296.

[26] Yeh D M, Li Y W, Chu W. Extracting entity relationship diagram from a table-based legacy database. *Journal of Systems and Software*, 2008, 81(5): 764-771.

[27] Strobl S, Bernhart M, Grechenig T *et al.* Digging deep: Software reengineering supported by database reverse engineering of a system with 30+ years of legacy. In *Proc. the 25th IEEE Int. Conf. Software Maintenance*, September 2009, pp.407-410.

[28] Zhang Z P, Yang H J. Incubating services in legacy systems for architectural migration. In *Proc. the 11th Asia-Pacific Software Engineering Conf.*, November 2004, pp.196-203.

[29] Zhang Z P, Liu R M, Yang H J. Service identification and packaging in service oriented reengineering. In *Proc. the 17th International Conference on Software Engineering and Knowledge Engineering*, July 2005, pp.620-625.

[30] Chen F, Li S Y, Yang H J *et al.* Feature analysis for service-oriented reengineering. In *Proc. the 12th Asia-Pacific Software Engineering Conf.*, December 2005, pp.201-208.

[31] Guo H, Guo C Y, Chen F *et al.* Wrapping client-server application to Web services for Internet computing. In *Proc. the 6th Int. Conf. Parallel and Distributed Computing Applications and Technologies*, December 2005, pp.366-370.

[32] Li S, Tahvildari L. JComp: A reuse-driven componentization framework for Java applications. In *Proc. the 14th IEEE Int. Conf. Program Comprehension*, June 2006, pp.264-267.

[33] Cuadrado F, García B, Dueñas J C *et al.* A case study on software evolution towards service-oriented architecture. In *Proc. the 22nd Int. Conf. Advanced Information Networking and Applications-Workshops*, March 2008, pp.1399-1404.

[34] Marchetto A, Ricca F. Transforming a Java application in an equivalent web-services based application: Toward a tool supported stepwise approach. In *Proc. the 10th Int. Symp. Web Site Evolution*, October 2008, pp.27-36.

[35] Marchetto A, Ricca F. From objects to services: Toward a stepwise migration approach for Java applications. *Int. Journal on Software Tools for Technology Transfer*, 2009, 11(6): 427-440.

[36] Huang G, Liu T C, Mei H *et al.* Towards autonomic computing middleware via reflection. In *Proc. the 28th Annual Int. Computer Software and Applications Conference*, September 2004, pp.135-140.

[37] Huang G, Mei H, Yang F Q. Runtime software architecture based on reflective middleware. *Science in China Series F: Information Sciences*, 2004, 47(5): 555-576.

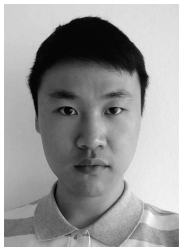
[38] Albertini B, Rigo S, Araujo G *et al.* A computational reflection mechanism to support platform debugging in SystemC. In *Proc. the 5th IEEE/ACM Int. Conf. Hardware/Software Codesign and System Synthesis*, September 2007, pp.81-86.

[39] Albertini B, Rigo S, Araujo G. Computational reflection and its application to platform verification. *Design Automation for Embedded Systems*, 2012, 16(1): 1-17.

[40] López P G, Fernández-Casado E, Angles C *et al.* Enabling collaboration transparency with computational reflection. In *Proc. the 16th Int. Conf. Collaboration and Technology*, September 2010, pp.249-264.

[41] Bellman K L, Nelson P R, Landauer C. Active experimentation and computational reflection for design and testing of cyber-physical systems. In *Proc. the Poster Workshop at the 2014 Complex Systems Design & Management International Conference Co-Located with the 5th International Conference on Complex System Design & Management*, November 2014, pp.251-262.

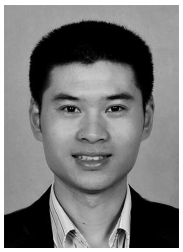
[42] Demers F N, Malenfant J. Reflection in logic, functional and object-oriented programming: A short comparative study. In *Proc. the IJCAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI*, August 1995, pp.29-38.



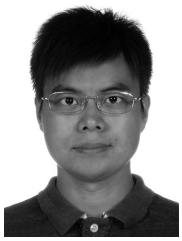
Su Zhang is a Ph.D. candidate in Key Laboratory of High-Confidence Software Technology and School of Electronics Engineering and Computer Science, Peking University, Beijing. His current research interests include system software and software engineering.

Hua-Qian Cai is now a postdoctoral researcher in Peking University, Beijing. He received his Ph.D. degree in computer software and theory from Peking University, Beijing, in 2018. His current research interests include programming language and software engineering.

Yun Ma is now a postdoctoral researcher in School of Software, Tsinghua University, Beijing. He received his Ph.D. degree in computer software and theory from Peking University, Beijing, in 2017. His current research interests include Web engineering and mobile computing.



Tian-Yue Fan is a senior manager and the intelligent manufacturing manager in Project Implementation Department of Hengyi Petrochemicals CO., LTD., Hangzhou. His research interests include intelligent manufacturing, industrial big data and informatization of business management.



Ying Zhang is an assistant professor at Peking University, Beijing. He received his Ph.D. degree in computer software and theory from Peking University, Beijing, in 2012. His current research interests include mobile computing and cloud computing.

Gang Huang is a full professor at Peking University, Beijing. He received his Ph.D. degree in computer software and theory from Peking University, Beijing, in 2003. His current research interests include operating systems, cloud computing, and Internetware.