A Proactive Support for Conceptual Interoperability Analysis in Software Systems

Hadil Abukwaik, Dieter Rombach
Computer Science Department
Kaiserslautern University
Kaiserslautern, Germany
{abukwaik, rombach}@cs.uni-kl.de

Matthias Naab
Fraunhofer IESE
Kaiserslautern, Germany
matthias.naab@iese.fraunhofer.de

***Abstract*—Successfully integrating a software system with an existing other software system requires, beyond technical mismatches, identifying and resolving conceptual mismatches that might result in worthless integration and costly rework. Often, not all relevant architectural information about the system to integrate with is publicly available, as it is hidden in internal architectural documents and not exposed in the public API documentation. Thus, we propose a framework of conceptual interoperability information and a formalization of it. Based on this framework, a system's architect can semi-automatically extract interoperability-relevant parts from his architecture and lower-level design documentation and publish it in a standardized and formalized way. The goal is to keep the additional effort for providing the interoperability-relevant information as low as possible and to encourage architects to provide it proactively. Thus, we extract from UML diagrams and textual documentation information that is relevant for conceptual interoperability. Companies that aim at interoperation of their systems with others, e.g. companies initiating an ecosystem, should be highly motivated to provide such interoperability information in order to grow their business impact by more successful interoperations. In a more advanced level, also the architect, who is integrating his system with a provided one, could extract interoperability-related information about his existing system and we envision to automatically match the pieces of both sides and identify conceptual mismatches.**

Keywords—conceptual interoperability; interoperability analysis; blackbox interoperation; information extraction; conceptual mismatches

I. INTRODUCTION

Software interoperability [1] importance is growing with the increasing attention given to interoperability-based systems like systems-of-systems, cyber-physical systems, and ecosystems. Building successful interoperation between software systems requires not only resolving their technical mismatches, but also the conceptual ones. Technical mismatches (e.g., different communication protocols, data types, arguments order, etc.) hinder exchanging information and services among the software systems. While, conceptual mismatches (e.g., architectural constraints, semantics, usage context, desired qualities, etc.) can produce meaningless or incorrect results that impose expensive rework even after spending effort on resolving technical mismatches.

Based on a scoping study we performed earlier [2], current interoperability approaches lack the focus on the conceptual aspects of interoperating software systems. The widely adopted strategy for interoperability analysis is performed by software clients through reading the externally shared Application Programming Interface (API) documentation of the system of interest in order to find its technical constraints [3]. However, this strategy is impractical for a software architect who is responsible for extracting the conceptual constraints of the external system to assess its conceptual mismatches with his system. From one hand, the API documentation (1) lacks sharing conceptual constraints which mainly exist in the unshared architectural documents (e.g., UML structure diagram shows data inherited constraints from generalization relations), (2) has technical information which is irrelevant to the conceptual analysis (e.g., the used programming language), and (3) has implicit assumptions that may be indirectly expressed (e.g., conditional sentences denoting usage context). From another hand, it has been shown [4] that manual, unguided investigation of API documentation is tedious and time consuming even for identifying one type of conceptual constraints. In fact, a study [5] showed that interoperation constraints are still being slipped despite of the software documentation familiarity. This is because of the verbose text of documentations and their informality that requires manual linguistic analysis skills for capturing, structuring, and saving the constraints information.

In this paper, we propose a COnceptual INteroperability Analysis (COINA) framework that aims at assisting software architects in proactively publishing the COnceptual Interoperability coNstraints (COINs) about their software systems while keeping the associated effort as low as possible. One of the key contributions of our work is the semi-automatic extraction of the system's COINs from hits (1) internally shared artifacts (UML documentations and architectural scenarios), and (2) externally shared API documentation. Our proposed framework automatically documents the extracted COINs in a standard formal document called the "COINs Portfolio" which is a published artifact of the software system. We expect COINA to increase the software architects' effectiveness and efficiency in identifying the conceptual constraints of their software systems. We also envision to automatically match the formal COINs Portfolios' of two software systems to identify their conceptual mismatches and to estimate required integration effort. This can stretch the trade-off space for selecting the most appropriate software system to interoperate with. Our framework is particularly

appropriate for software companies interested in building systems that interoperate with others (e.g., initiators of ecosystems). From long-term perspective, companies providing their conceptual interoperability information will grow their business impact and competitiveness by providing high interoperation-success likelihood while keeping low integration effort. Our approach is tool supported and we plan to evaluate it through a controlled experiment.

The rest of the paper is organized as follows: In Section II we review related works. In Section III and IV, we describe our COINA framework and its supportive tool respectively. We discuss COINA limitations in Section V, while we conclude and present our future work in Section VI.

II. Related Work

A. Extracting conceptual constraints in black-box context

Mining API documentation. API documentation is a valuable resource of information especially in black-box interoperation. Accordingly, many static approaches have been proposed to mine conceptual information from them. From API documentation, Wu et al. [4] infer dependency constraints among parameters, Pandita et al. [6] infer methods' pre and post conditions, and Zhong et al. [7] infer resource specifications. Dekel and Herbsleb [8] extract method rules from API documentation and push them into programming editors. Our COINA framework complements these approaches as follows. First, COINA infers conceptual constraints for both services and exchanged data to enrich the input for the conceptual analysis with more interoperability information and consequently increase the analysis effectiveness. Second, most of the preceding approaches apply NLP with rule-based techniques, but COINA combines these with machine learning (ML) to improve the recall and precision results.

Mining software executions. There are also dynamic approaches for extracting software constraints from its execution. Through running test suites, Nimmer and Ernst [9] infer software invariants, while Gabel and Su [10] infer temporal constraints. Gao et al. [11] infer data preconditions for web services from API signatures, error messages, and testing results to increase the confidence in the mined constraints. Betrolino et al. [12], learn a service behavioral protocol by observing its execution. Results' accuracy in these approaches depends in part on the quality and completeness of the designed test cases. Also, analyzed executions have to fully characterize all possible executions of the service to be credible. To increase the confidence in the mined information and consequently the increase the correctness of the conceptual interoperability analysis, COINA proposes an in-house extraction for the interoperability constraints from both shared API documentations and unshared architecture scenarios and UML documentations. This also lessens the individual constraints-inference cost spent by each third-party system and raises the software value and competitiveness.

B. Conceptual mismatch detection in black-box context

Reuse and COT analysis. Many approaches have been proposed to address the issue of finding mismatches between components. For example, Bhuta et al. [13] propose to manually create component definitions for technical and architectural assumptions, then to apply rules on them to detect the mismatches. COINA extends the architectural part of this work by covering more conceptual constraints (e.g., context and quality constraints) to eliminate the risks resulted from unplanned conceptual mismatches. Also, instead of writing a component's definition from scratch in a completely manual manner, we first try to benefit from already existing software documentation by automatically collecting as much as available interoperability constraints from them. Then, the human takes place in completing the task as needed. We expect our approach to reduce the effort for identifying the interoperability constraints. Testing-based techniques like [3] are useful in detecting software mismatches. However, they need creating complete, high-quality test suites and launching the interoperation for each test. Besides, intensive testing is not always feasible due to invocation costs. Also, some methods consider prototyping to evaluate the components by simulating its usage within other systems [14]. As this requires component acquisition, learning, and evaluating, it is expensive and limits the number of analyzed components.

Conformance checking. This method has an active research field with techniques for enforcing design-by-contract [15]. These techniques allow functions' authors to formally specify their methods' pre and post conditions, and object invariants to perform static [5] or dynamic conformance analysis [3]. Although these techniques are useful in detecting conformance violations automatically, they focus on technical constraints rather than conceptual ones. For instance, proposed contracts in [5] are designed to check null value, value ranges, and object size. Therefore, COINA meets the need for a complementary approach that stretch the conformance checking to the conceptual level while keeping the effort of extracting and formalizing the conceptual constraints low to make the approach more practical and usable.

III. Conceptual Interoperability Analysis (COINA) Framework

Our framework includes two key components: the COINs Portfolio and its Generator. The inputs to our framework are the software system's already existing architectural documentation (UML diagrams and architectural scenarios), its API documentation, and its list of interoperability elements (a manually created list by the software architect which includes the names of all data items and services involved in the software system interoperation). The output is a standard formal document explicitly showing the system's conceptual interoperability constraints (COINs Portfolio). The remainder of this section presents the framework components in details.

A. Conceptual Interoperability Constraints (COINs) Portfolio

We define the COINs of a software system as the high-level characteristics governing its interoperability, i.e., any misassumption in them may lead to conceptually-wrong, meaningless, or improper interoperation results. Explicitly declaring the COINs of software systems facilitates detecting their conceptual mismatches and allows better planning for the resolution. The set of COINs we introduce in this paper are derived from various literature sources. We selected them based on a criterion: representing a source of conceptual interoperability mismatch. Table I presents the current set of

COINs and their categories with examples. **Syntax COINs** specify the concept-packaging methods (i.e., the conceptual modeling language) and the lexical references used in the system. Examining the syntactic match paves the way towards investigating the semantic one. **Semantic COINs** state semantic constraints (e.g., the measurement unit of a *calculateDistance* service is km not mile), and semantic references (e.g., reference ontologies) that encode the meaning of exchanged data and service goals. As no reference ontology has been widely adopted yet, we consider this a theoretical constraint which is left for future advances in the ontology research. **Structure COINs** depict system's elements, their relations, and their arrangements that influence the interoperation results, e.g., interoperating with a software system without being aware of its data distribution may introduce a security threat if network links between remote sites are not encrypted. In this case, the distribution of the system is a structural COIN. **Dynamic COINs** report information about interoperability elements' behavior during interaction that if missed, it can introduce conceptual flaws. For example, interoperating with a system of regularly changing data may lead to synchronization issues. **Context COINs** pertain to external aspects forming the interoperation settings, i.e., user and usage properties. For example, software systems designed to interoperate with desktop devices may cause display and memory issues for mobile devices. **Quality COINs** capture required and provided quality characteristics related to exchanged data and services. For example, inaccurate results may occur when interoperating with a face detection service that requires a specific resolution for the input image. In the proposed framework, all instances of the previous COINs, which are related to interoperability elements, should be documented in the system COINs Portfolio. As seen in Fig. 1, each COIN instance in the Portfolio has a dedicated sheet with fields reporting its details (e.g., category, name, value, weight, consequences, etc.).

TABLE I. CONCEPTUAL INTEROPERABILITY CONSTRAINTS

Category	COIN name	Examples of value
Syntax	Lexical references	Dictionary, thesaurus, glossary, etc.
	Modeling lang.	XML, UML, ADL, WSDL, etc.
Semantic	Semantic references	Reference ontologies
	Semantic constraints	Data units and scale ratio
Structure	Data structural constraints	Invariants, inherited constraints, and multiplicity constraints
	Data storage	Relational database, flat files, etc.
	Distribution	Distributed, centralized
	Encapsulation	Encapsulated , not encapsulated
	Concurrency	Single-threaded, multi-threaded
	Layering	Layered, not layered
Dynamic	Data change	Periodic, irregular, continuous, etc.
	Service conditions	Pre, post, and time conditions
	Interaction properties	State(ful/less), (a)synchronous, etc.
	Interaction time constraints	Session timeline, acknowledgment timeline, response timeline, etc.
	Communication style	Messaging, procedure call, blackboard, streaming
Context	Usage context	device type, wired/wireless, access rate, time, location, etc.
	Intended users	Human/machine, gender, age, etc.
Quality	Data quality	Security, trust, accuracy, etc.
	Service quality	Safety, availability, efficiency, etc.

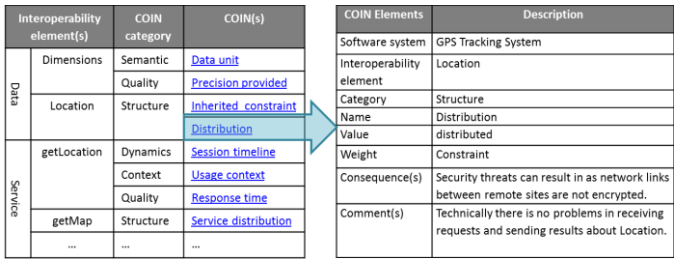

Fig. 1. Example of a COINs Portfolio (left) and one of its sheets (right)

B. *COINs Portfolio Generator*

Manual creation of the COINs Portfolio and its sheets is a cumbersome and expensive task as it requires sifting through the software documentation to get its conceptual interoperability constraints. Hence, we support software architects in this task by the Portfolio Generator that semi-automatically creates the portfolio.

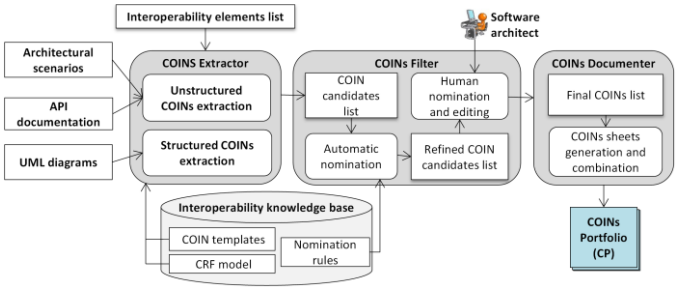

Fig. 2. Overview of the COINs Portfolio Generator.

The Generator consists of the following subcomponents (see Fig. 2). **The Interoperability Knowledge Base (IKB)** is a repository of our predefined, reusable knowledge that gets accessed by the other subcomponents of the Generator to perform their activities. In a software company, the IKB can be maintained and updated by its software architects. The first step towards building the Portfolio starts with **the COINs Extractor**, which generates a COIN candidates list through two activities: (1) ***Structured COINs extraction*** takes the UML diagrams as an input and checks them against the predefined COIN templates in the IKB. These templates are rules, which if satisfied a COIN candidate is extracted and added to the candidate lists. For example, if a database *LocationDB* is deployed on many nodes, the Extractor will add a candidate COIN instance for the *Location* interoperability element (see Fig. 1), where category = "Structure", name = "Distribution", value = "distributed", and weight = "constraint". ***(2) Unstructured COINs extraction*** takes the natural text in the architectural scenarios and API documents and parses it using a Natural Language Processing (NLP) technique called dependency parsing [16]. Then, the parsed text gets checked against our Conditional Random Field (CRF) model [17] which is trained and saved in the IKB. For instance, if a conditional sentence has in its action clause a noun of a value equal to a service name in the interoperability elements list (e.g.,"*If* Client_account_type = premium, the *getLocation* service is open"), the Extractor will add a candidate COIN instance for the *getLocation interoperability element*, where category = "Dynamic" , name = "Service condition", value =

"Client account = premium", and weight = "constraints". Note that consequences and comments fields of the COIN cannot be automatically extracted, but the architect can add them later in the final Portfolio if needed. When extraction is done, **the COINs Filter** runs a first automatic nomination for the COIN candidates based on our predefined nomination rules that we saved in the IKB. For example, a redundancy elimination rule removes duplicates that are extracted from two sources like a UML diagram and the API documentation. Then, a second nomination is performed by the software architect who reviews the refined COIN candidates list to add, delete, and edit COINs as needed. **The COINs Documenter**, finally, creates the standard sheet for each COIN in the final COIN list and combines them in the COINs Portfolio which can always be updated by the architect. In practice, COINs Portfolios would be shared online or saved in a Portfolios Repository that a third-party vendor could maintain and provide licensed access. As formalizing the Portfolios would automate their conceptual interoperability analysis, our undergoing research is extending the framework with Portfolios' Formalizer and Mapper components that we plan to present in a future publication.

IV. Tool Support

To technically support our framework, we developed the first version of our Conceptual Interoperability Analyzer Tool - the Portfolio Generator. We built it as an add-in for the Enterprise Architect [18]. Fig. 3 shows an example of its structured extraction results. A new version of the tool is under construction to include the unstructured COINs extraction.

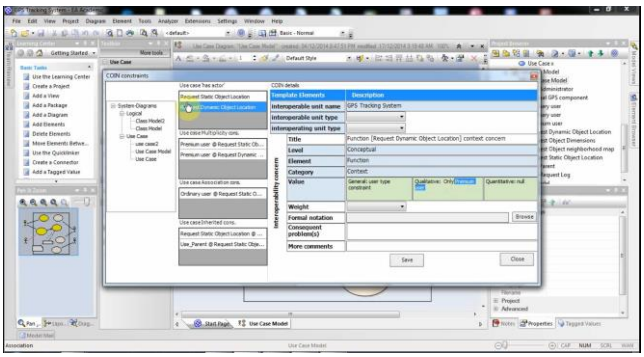

Fig. 3. Example of the extracted COINs in the Extractor tool.

V. Framework Limitations

COINA is subjective to the quality of the assumed-to-exist architectural documentations, e.g., missing constraints in the UML diagrams will be missed during the structured extraction. To mitigate this issue, we further include the API documentation to complement the extraction results. Also, false negatives occur if a COIN instance is not addressed by one of the predefined COIN templates. To alleviate this issue, the framework modularity allows extending its IKB with more COIN templates as needed. Moreover, the friendly interfaces of our tool, reduces the amount of added effort for creating the list of interoperability elements and for filtering the COINs.

VI. Conclusion and Future Work

This paper presents a framework for supporting conceptual interoperability analysis. The framework semi-automatically extracts the conceptual interoperability constraints of a software system from its existing architectural and API documentations. As an output, it creates a standard COINs Portfolio, which is expected to increase the conceptual interoperability analysis efficiency and effectiveness especially when it gets formalized. Currently, we have developed the UML COINs templates, their extraction algorithms, and the nomination rules. We have also implemented the first version of a supportive tool (the Portfolio Generator). Our work plan includes: extending COINA with Portfolios' Formalizer and Mapper components, improving the tool, and evaluating the framework through a controlled experiment.

Acknowledgment

This work is performed as part of the PhD research of Hadil Abukwaik under the supervision of Prof. Dieter Rombach. We thank Dr. Jörg Dörr for his insightful discussions and guidance.

References

[1] A. Geraci, IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries, 1991, p. 114.

[2] H. Abukwaik, D. Taibi and D. Rombach, "Interoperability-related architectural problems and solutions in information systems: A scoping study," in *ECSA*, 2014.

[3] S. Hallé, et al., "Runtime verification of web service interface contracts," *Computer*, vol. 43, no. 3, pp. 59--66, 2010.

[4] Q. Wu, et al., "Inferring dependency constraints on parameters for web services," in *22nd Inter'l Conf. World Wide Web*, 2013.

[5] B. Rubinger and T. Bultan, "Contracting the Facebook API," in *4th Inter'l Workshop on Testing, Analysis and Verification of Web Software*, vol. 35, 2010, pp. 63-74.

[6] R. Pandita, et al., "Inferring method specifications from natural language API descriptions," in *ICSE*, 2012, pp. 815--825.

[7] H. Zhong, et al., "Inferring resource specifications from natural language API documentation," in *ASE*, 2009, pp. 307--318.

[8] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *ICSE*, 2009.

[9] J. W. Nimmer and M. D. Ernst, "Automatic Generation of Program Specifications," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 229--239, 2002.

[10] M. Gabel and Z. Su, "Testing mined specifications," in *FSE*, 2012, pp. 4:1--4:11.

[11] C. Gao, J. Wei, H. Zhong and T. Huang, "Inferring Data Contract for Web-Based API," in *IEEE Inte'l Conf. Web Services (ICWS)*, 2014, pp. 65--72.

[12] A. Bertolino, P. Inverardi, P. Pelliccione and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in *ESEC/SIGSOFT FSE*, 2009, pp. 141--150.

[13] J. Bhuta, C. A. Mattmann , N. Medvidovic and B. Boehm, "A framework for the assessment and selection of software components and connectors in cots-based architectures," in *WICSA*, 2007, pp. 6--6.

[14] R. Land, L. Blankers, M. Chaudron and I. Crnković, "COTS selection best practices in literature and in industry," in *ICSR*, 2008, pp. 100--111.

[15] B. Meyer, "Applying'design by contract'," *Computer*, vol. 25, no. 10, pp. 40--51, 1992.

[16] M. A. Covington, "A fundamental algorithm for dependency parsing," in *The 39th Annual ACM Southeast Conf.*, 2001, pp. 95-102.

[17] J. D. Lafferty, A. McCallum and F. C. N. Pereira, "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data," in *18th Inte'l Conf. Machine Learning*, 2001, pp. 282-289.

[18] S. Systems, "Enterprise Architect," [Online]. Available: http://sparxsystems.de. [Accessed 19 November 2014].