Tool-supported extraction of conceptual interoperability constraints of software units from UML Diagrams

Hadil Abukwaik
Department of Computer Science
University of Kaiserslautern
Kaiserslautern Germany
Email: abukwaik@cs.uni-kl.de

Dieter Rombach
Department of Computer Science
University of Kaiserslautern
Kaiserslautern Germany
Email: rombach@cs.uni-kl.de

***Abstract*—Successfully integrating an external software unit into a system requires software architects to check the conceptual constraints of this unit to ensure that it has no mismatches with the system. However, such constraints about software units are usually hidden within their architectural documents that are not publicly shared with clients. Hence, owners of the software units need to search for the constraints in the architecture document and provide them to clients. However, this manual task is not trivial and it is time consuming especially in the case of large software systems. In this paper, we demonstrate a tool-supported, systematic approach for extracting the interoperability-related constraints of software systems from their architecture and lower-level design documentation. Our proposed approach aims at helping architects in performing the conceptual interoperability analysis tasks in an effective and efficient manner. To bring the approach into practice, we developed an aiding tool that assists architects with easy-to-use interfaces. We plan to evaluate our approach empirically through a controlled experiment where we will test our hypotheses about its positive effect on architects effectiveness and efficiency in performing the interoperability analysis.**

Keywords- conceptual interoperability; interoperability analysis; information extraction; software architecture; tool support

I. Introduction

Interoperability is the ability of two or more separately-developed software units to communicate and exchange data in a seamless and meaningful way [1]. This ia an important property as it plays a vital role in today's large software systems. For example, a system-of-systems [2] needs interoperability to put together a number of information systems in order to achieve more functionality and better performance (e.g., a health care system-of-systems that combines application of doctor, patient, and pharmacy for better treatment). Similarly, cyber-physical systems [3] need interoperability to link computational software units with physical resources (e.g., an automated traffic control cyber-physical system that connects traffic lights and vehicles on the road). Having this being said, to successfully integrate software units, it is required to identify and resolve any technical or conceptual interoperability mismatches. While technical mismatches (e.g., different communication protocols, programming languages, data types, arguments orders, etc.) hinder the exchange of information and services among software units; conceptual mismatches (e.g., usage context, architectural constraints, terminologies, qualities, etc.) lead to meaningless or undesirable interoperation results. Such conceptual mismatches are expensive to resolve and should be found early in the integration project before spending effort on resolving the technical mismatches.

According to a scoping study we performed earlier [4], existing works focus mainly on achieving the technical level of interoperability of software units and lack the focus on the conceptual level. In the context of black-box interoperations, in which the source code of a software unit is not publicly shared, the typically available source of information about a software unit is the documentation of its Application Programming Interface (API). This API documentation describes the software unit in terms of its functionality, input, output, and data type. Hence, interested clients in a software unit investigate its API documentation to find the technical constraints [5].

However, API documentation of a software unit is insufficient source of information for a software architect who is assessing the conceptual interoperability mismatches. This is because API documentations are typically technical oriented and do not expose all conceptual and architectural constraints that are usually hidden in the unshared architectural documents. These architecture documents are usually written using the popular Unified Modeling Language (UML) [6] that enables describing the software structure (e.g., component diagram) and behavior (e.g., state chart diagram). On another hand, it is a tedious and time consuming task for the owners of a software unit to manually find the interoperability-relevant pieces of information about a specific software unit from across the whole system's UML diagrams, and then to add these information to the shared API documentation for interested clients.

In this paper, we present a tool-supported approach for extracting the **CO**nceptual **I**nteroperability co**N**straints (COINs) from UML diagrams. This work is an extension of our previously proposed framework for supporting interoperability analysis [7]. The goal of our presented COINs' extraction

approach is to alleviate the burden from software architects' shoulders in explicitly publishing the conceptual information about their interoperable software units, and accordingly to support interested clients in performing proper interoperability analysis in order to effectively detect the conceptual mismatches. This is achieved through the: (1) semi-automatic extraction of COINs from internal UML documentation, and (2) automatic documentation of extracted COINs in a standard ready-to-publish document that we call the "COINs Portfolio". Our work offers advantages for software companies that build interoperable systems. That is, on the short-term planning, we expect that the proposed extraction idea would help these companies increasing their effectiveness and efficiency in identifying and documenting the conceptual constraints of their interoperable software systems. Besides, on the long-term planning, these companies would grow their business value as a result of publishing sufficient interoperability information about their systems that would lead to more successful interoperations with less integration effort.

The rest of the paper is organized as follows: We start with a motivation scenario in Section II. Then, we describe our COINs extraction approach in Section III. In Section IV and V, we discuss the planned evaluation for our work and related work accordingly. We finally conclude and present future work in Section VI.

II. Motivating Example

This section outlines a brief example for the application of our tool-supported COINs extraction approach.

Imagine a company *Alpha* that has developed an *ATM (S1)*, which is a cash machine developed with the goal to make its software system interoperable. That is, *S1* is intended to exchange data and services in a meaningful way with other separately developed software systems (e.g., mobile apps of bank clients, bank management systems, etc.). Hence, the company has created the API documentation of *S1* and published it for interested third-party clients.

Afterwards, another company *Beta* gets interested in integrating an instance of *S1* within their *Bank system (S2)*. The responsible person for assessing the feasibility of building a successful interoperation in this scenario is the software architect of *S2*, Noah. He is responsible for analyzing the interoperability constraints for both *S1* and *S2* and he aims at detecting any conceptual mismatches between them. Hence, Noah starts with a manual investigation of the text in *S1*'s API documentation for its offered functionalities and related data (e.g., "Withdrawal" function and its related data element "Amount" that represents the requested money to be withdrawn). Based on this investigation, Noah could identify some technical mismatches, for example, he finds that *S1* uses the Secure Socket Layer (SSL [8]) protocol to ensure the security over communications, while *S2* uses another security protocol called the Transport Layer Security (TSL [9]). However, Noah could not detect some of the conceptual mismatches because their related constraints are hidden in the unshared UML diagrams of *S1*. Here are some explaining examples:

- A class diagram of *S1* shows that the "Withdrawal" transaction has a "history" file that contains up to 100 transactions, while *S2* assumes that the "history" contains all withdrawal transactions on the account within a year (which may be more than 100);
- A use case diagram of *S1* states that the "Withdrawal" transaction is permitted only for the owner of the account, while *S2* allows shared account for a husband and wife where some mechanism is used to handle simultaneous access);
- A sequence diagram of *S1* shows that the interaction type with the "Withdrawal" functionality is synchronous meaning that the user can make a second withdrawal transaction before receiving a confirmation on the first. While, *S2* uses asynchronous interactions that require a confirmation on a transaction before allowing the second one.

Obviously, having such information in the API document would enhance the results of Noah's analysis and would save the cost spent on handling the unexpected conceptual mismatches late in the integration project.

A possible approach to handle this issue is to contact company X asking for further information about *S1*. However, adopting this solution might be expensive for company X especially for repeated inquiries. Also, the waiting time might be inconvenient for clients. Alternatively, we propose that company X publishes the relevant conceptual information of *S1* in advance for interested clients. To make this practical and avoid the manual, time-consuming effort , especially for large systems, we support architects with a tool that semi-automate the extraction and documentation of the conceptual constraints of their interoperable systems!

III. Tool-supported Extraction of Conceptual Interoperability Constraints

This section describes our COIN model, the extraction process, tool support, and planned evaluation.

A. The Conceptual Interoperability Constraint (COIN)

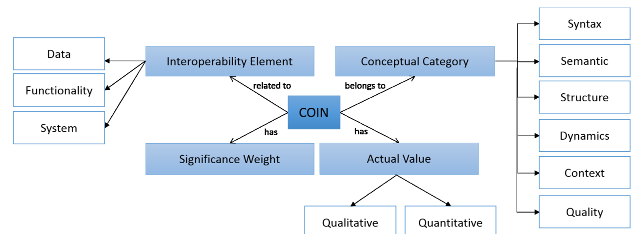

Fig. 1. A COIN meta-model.

The corner stone of our work is the **COINs** or the conceptual interoperability constraints, which we define as the conceptual characteristics of the software unit that if they are misassumed, they may lead to conceptually-wrong, meaningless, or improper interoperation results [7]. In Figure 1, we present our COIN meta-model and show its structure. Each

COIN in the interoperable software system is related to an element (i.e., data, function, component, or system), belongs to a conceptual category (i.e., syntax, semantic, structure, dynamic, context, and quality), has a value (i.e., qualitative and/or quantitative description), and has a significance weight (i.e., high, medium, or low).

The COINs we target in our extraction approach are the ones that represent a cause of conceptual interoperability mismatch between two software systems. Figure 2 presents the whole set of COINs and their categories with examples. Note that, the shaded rows outlines the so far supported COINs with the current version of our extraction tool.

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. 2. Conceptual interoperability constraints (COINs).

B. *Extracting COINs from UML Diagrams*

As we mentioned in Section I and II, creating the COINs Portfolio manually is a tedious and expensive task. This is beacause it requires sifting through the UML documentation of the whole software system, and then extracting the only useful pieces of information for interoperability analysis. To address this issue, we have previously proposed an abstract idea about a Portfolio Generator [7]. In this subsection and the one after we describe this idea in details along with its supporting tool.

In our approach, we aim at gathering all related COINs of an interoperable system into a standard document called the "COINs Portfolio". The input to our COINs' extraction approach is a consistent, complete, and up-to-date UML document about the interoperable software unit. This UML document includes structural diagrams (e.g., component diagram, deployment diagram, class diagram, etc.) and behavioral diagrams (e.g., use case diagram, sequence diagram, etc.). This input goes through the following four activities in order to result in the desired COINs Portfolio (as seen in Figure 3):

Identification of interoperable elements. In this activity, the software architect identifies the UML elements (i.e., components, classes, use cases, or actors) of the system, which are involved in interoperations with other software systems. This identification happens in terms of assigning an "Interoperability Type" property for the element. For example, in the ATM example described in Section II, the "Withdrawal" use case in the use case diagram would have the interoperability property declared, and similarly this would be declared for the the "Amount" data element in the class diagram. Declaring this property for the elements directs the subsequent activities of the COINs extraction.

Automatic extraction of COINs. To enable the proposed automation in this activity, an Interoperability Knowledge Base (IKB) is created and charged with COIN extraction templates from UML diagrams. A COIN extraction template is a set of rules that if it is met by an interoperable element within a UML diagram, then a COIN candidate for the element is created and added to the list of COIN candidates. An example of a context COIN template is:

- **if** there is a use case within a use case diagram that is identified as interoperable element (e.g., the "Withdrawal" use case within the use case diagram of *S1*),
- **and** it is inherited from another use case (e.g., "Withdrawal" use case is one kind of the "Transaction" use case,
- **and** the inherited from use case has a constraint (e.g., the "Transaction" use case has a constraint that only the account owner can do it),
- **then**, a context COIN will be added to the list of COIN candidates for the use case element (e.g., the "Withdrawal" use case has a constraint that only the account owner can do it.

Note that the extraction activity starts with checking each UML diagram of the systems to find if it contains any element with the interoperability property declared. Then, only the interoperable elements are checked against the predefined COIN templates saved in the IKB.

Manual filtering for the extracted COIN candidates. This activity is performed manually by the system architects who have the final word to approve or disapprove the automatically extracted COINs within the final published COINs Portfolio. Furthermore, they can manually add more COINs to the Portfolio if they see it important and useful to share with clients.

Automatic generation for the COINs Portfolio. Finally, the approved and the manually added COINs are bundled together and categorized according to the elements they are related to. Then, they are documented in a standard form that is ready to share with clients who will use it for their interoperability analysis task if interested in interoperating with the software system.

C. *The CoinsExtractor Tool*

To make our proposed idea applicable in practice, we implemented the CoinsExtractor tool [10] that assists architects with easy-to-use interfaces in extracting and publishing the COINs of the interoperable units of their software systems. This section summarizes the main features, design, and limitations of the tool (for details please refer to [10]).

Main features. The CoinsExtractor has a number of features that supports architects through the aforementioned ac-

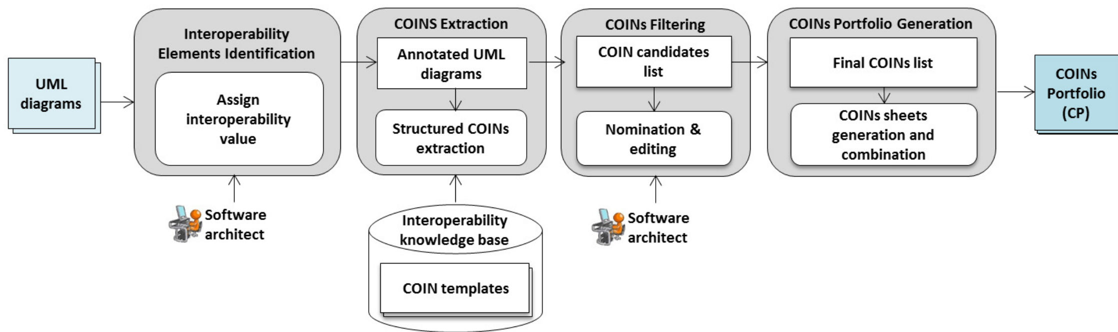

Fig. 3. COINs extraction process.

tivities in Subsection B, which takes some burden off the architects' shoulders as the following:

- *Interoperability Tags*. With these self-implemented tags for the UML elements, the architect can directly determine their systems' interoperable elements. Accordingly, if an element is tagged as interoperable, all its instances within all the diagrams get tagged as interoperable too. The tool maintains a table of interoperable elements, which represents the architect knowledge about the interoperable units of the system and it would be reused in all future integration projects.
- *COINs Extraction*. This feature implements the COIN templates that we described earlier in Subsection B. Hence, the tool saves the architect's time and effort by automatically looking for the relevant COINs about the interoperable elements only. In addition, this feature guarantees consistency in terms of what COINs are being extracted and how they are documented across projects of interoperable units or systems.
- *COINs Review*. The CoinsExtractor tool realize the COINs filtering by enabling the architects to efficiently review, update, approve, or delete the automatically extracted COINs. It offers two views: (1) *diagram-based view* where COINs can be navigated according to the source diagram that they are found in, and (2) *element-based view* where COINs can be navigated according to the the interoperable element they belong to.
- *COINs Portfolio Generation*. Finally, the tool creates a ready-to-share, web-based COINs Portfolio (see an example in Figure 4). The Portfolio arranges the COINs according to their categories to facilitate the conceptual interoperability analysis task that will be performed by interested clients.

Design and Implementation. The CoinsExtractor has a multi-layered architecture with a *presentation layer* that is responsible for tool-architect interactions, *business layer* that includes the logic units responsible for processing the UML diagrams and extracting their COINs, the *data access layer* that reads input from the UML database and writes results into the output file, and finally the IKB that stores all predefined COIN templates as we described in the previous subsection. These layers lead to the tool modularity that allows extending its COIN templates to cover more categories and accordingly enhancing the interoperability analysis results. The CoinsExtractor tool is implemented as an extension for the Enterprise Architect (EA) application, which is one of the powerful, widely-used architecture modeling tools [11].

Limitations. The current version of the tool accepts input (i.e., UML diagrams) created by the Enterprise Architect only. It also assumes that the UML input is created according to the UML standard notations specified by the Object Management Group (OMG) [12] and published by the International Organization for Standardization (ISO) [13]. To ensure that these assumptions are realistic enough to use the tool in industry, we have reviewed the tool with architect experts and we used their feedback to enhance the tool usability.

COINs Portfolio

file:///C:/Users/abukwaik/Desktop/Exported.html

COINs Portfolio

Interoperability element(s)		COIN Category	COIN Title	COIN sheet (details)
Element	Object			
Data	Activity Log	Structure	Data [Activity Log] structure constraint	click here
		Structure NL	Data [Activity Log] dynamic constraint	click here
	Night Driving	Dynamic NL	Data [Night Driving] structure constraint	click here
	Request Response	Structure NL	Data [Request Response] dynamic constraint	click here
	Steering	Dynamic NL	Data [Steering] dynamic constraint	click here
Function	Auto Steering	Context	Function [Auto Steering] context constraint	click here
		Structure	Function [Auto Steering] structure constraint	click here
	Remote Steering	Context	Function [Remote Steering] context constraint	click here
		Structure	Function [Remote Steering] structure constraint	click here

Fig. 4. COINs Portfolio generated by CoinsExtractor [10].

IV. Planned Empirical Evaluation

To empirically evaluate the ideas that we have presented in this paper, we plan for conducting a controlled experiment. The goal of this experiment, formulated by means of the Goal-Question-Metric template [14], is to *analyze* the tool-supported approach for extracting COINs from UML diagrams *for the purpose of* evaluation *with a focus on* effectiveness, efficiency, and acceptance *from the perspective of* software architects *in the context of* a controlled experiment with students. For more

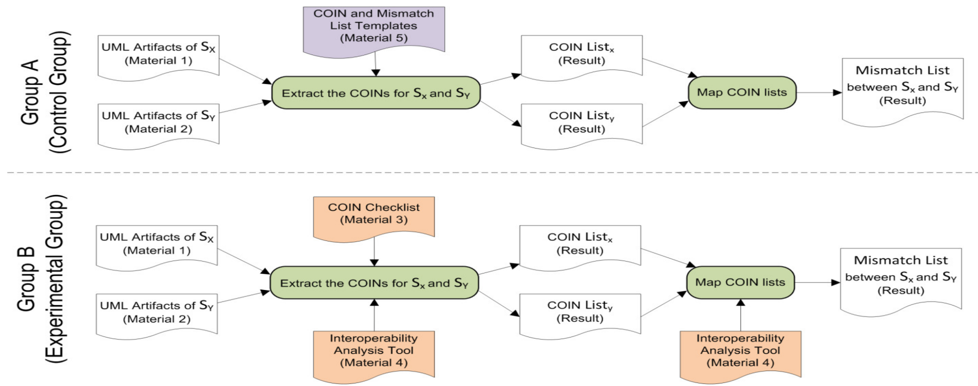

Fig. 5. Experimental design for evaluating the analysis approach.

details about the experiment (i.e., detailed information about the research questions, process, material, and data analysis plan) please refer to [15].

Figure 5 outlines the experimental design of the planned controlled experiment, in which we plan to have two groups of students who we hand them the same input (i.e., UML diagrams of two software systems) and we ask them to find the the COINs of each system in the first step and their mismatches in the second one. While, the control group applies ad-hoc approach in analyzing the input, the experimental group applies our proposed approach along with the support of our CoinsExtractor tool. Once we execute the experiment, we will analyze the collected data from both groups and compare between them with regards to correctness and completeness, and then we will publish the results.

V. Related Work

A. Detecting Mismatches in Black-box Context

Reuse analysis approaches have been proposed for finding component mismatches. Bhuta et al. [16] propose to manually create component definitions for technical and architectural assumptions, then to apply mismatch-detection rules on them. Our approach extends the architectural part of this work by further conceptual constraints to reduce the possibility of facing risks of unplanned conceptual mismatches. Besides, we try to reduce the cost by finding the already documented constraints in the UML with automation support, rather than writing definitions manually from scratch. **Testing-based techniques** such as Halle's [5] are useful in identifying mismatches, though, they require preparing complete, high-quality test suites and launching interoperation for each test case. Such intensive testing is expensive and impractical due to invocation costs. **Prototyping methods** for analyzing software components propose simulating its usage within other systems [17]; however, this may not always be feasible as it requires acquiring the component, learning, and evaluating it. **Conformance checking** has an active research community with design-by-contract enforcement techniques [18] where formal specification of methods pre-, post- conditions, and invariants are used in static [19] or dynamic [5] conformance analysis. Although this enables detecting technical conformance violations automatically, it does not address the conceptual ones. Therefore, our proposed idea stretches the conformance checking to the conceptual level.

B. Tools for Interoperation Analysis

Previous works have proposed tools to support the interoperability analysis task. For example, Bahuta et al. presented a tool called, the Integration Studio (iStudio) [16], which performs automatic assessment of architectures and proposes mismatch resolution. Compared to our CoinsExtractor tool, iStudio depends on a completely manual specification for the architectural interfaces, while our tool saves this cost through the semi-automatic extraction of such information from available UML documents. In addition, we extend the architectural attributes covered by iStudio with further conceptual features like context and quality. Another interoperability supporting tool was proposed by Ullberg et al. [20] for analyzing enterprise architecture models. This tool allows assessment theories to be specified using a formal modeling language (i.e., extension of the Object Constraint Language- OCL [21]), and then it utilizes it in the interoperability analysis. Relating this to our CoinsExtractor, our tool infers the conceptual features that would be the input for such assessment theories. On other hand, Buschle [22] developed a tool that analyzes properties including interoperability to provide decision-making support for information technology in enterprise architecture models. To the best of our knowledge, the CoinsExtractor is the first tool to semi-automate the extraction of conceptual interoperability constraints from UML diagrams, which aids architects and eliminates unexpected conceptual mismatches.

VI. Conclusion and Future Work

In this paper we have presented an approach for extracting the conceptual interoperability constraints of an interoperable software system from its UML diagrams. The approach aims at enhancing the effectiveness and efficiency of software architects in extracting and publishing their systems' COINs with third-part clients. We have also outlines the features of the supporting tool that we have implemented as an extension for the Enterprise Architect software application.

Based on our ongoing research, we aim to create further COIN templates to add them in our interoperability knowledge base, and to transform the COINs Portfolio into a formal notation to enable the automatic mapping of two interoperating systems to find their mismatches. We also plan to execute the designed controlled experiment and to evaluate the tool within industrial case studies too.

ACKNOWLEDGMENT

This work is part of the PhD research of Hadil Abukwaik under the supervision of Prof. Dieter Rombach and is funded by the PhD Program of Kaiserslautern University. The authors would like to thank Mohammed Abufouda, Shah Rukh Humayoun, and the anonymous reviewers for their constructive comments.

REFERENCES

[1] A. Geraci *et al.*, *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*, 1991.

[2] W. A. Crossley, “System of systems: An introduction of purdue university schools of engineering’s signature area.”

[3] R. Baheti and H. Gill, “Cyber-physical systems,” *The impact of control technology*, vol. 12, pp. 161–166, 2011.

[4] H. Abukwaik, D. Taibi, and D. Rombach, “Interoperability-related architectural problems and solutions in information systems: A scoping study,” in *Software Architecture*. Springer International Publishing, 2014, vol. 8627, pp. 308–323.

[5] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire, “Runtime verification of web service interface contracts,” *Computer*, vol. 43, no. 3, pp. 59–66, 2010.

[6] G. Booch, *The unified modeling language user guide*. Pearson Education India, 2005.

[7] H. Abukwaik, M. Naab, and D. Rombach, “A proactive support for conceptual interoperability analysis in software systems,” in *Working Conference on Software Architecture. 2015. WICSA’15*, 2015.

[8] A. Freier, P. Karlton, and P. Kocher, “The secure sockets layer (ssl) protocol version 3.0,” 2011.

[9] E. Rescorla, “The transport layer security (tls) protocol version 1.1,” *Transport*, 2006.

[10] H. Abukwaik, M. Abujayyab, and D. Rombach, “Coinsextractor: The architects’ buddy in identifying interoperability-relevant architectural constraints,” in *European Conference on Software Architecture. 2015. ECSA’15*, 2015.

[11] “Enterprise Architect,” http://sparxsystems.de/, Sparx System, accessed: 2015-05-08.

[12] O. CORBA and I. Specification, “Object management group,” *Joint revised submission OMG document orbos/99-02*, 1999.

[13] J. Koppell, “International organization for standardization,” *Handb. Transnatl. Gov. Inst. Innov*, vol. 41, no. 8, p. 289, 2011.

[14] V. R. Basili, “Software modeling and measurement: the goal/question/metric paradigm,” 1992.

[15] H. Abukwaik, “Empirical evaluation for the conceptual interoperability analysis approach- controlled experiment design,” Kaiserslautern University, CS Department, Tech. Rep., 2014. [Online]. Available: https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/3915

[16] J. Bhuta, “A framework for intelligent assessment and resolution of commercial-off-the-shelf product incompatibilites,” Ph.D. dissertation, University of Southern California, 2007.

[17] R. Land, L. Blankers, M. Chaudron, and I. Crnković, “Cots selection best practices in literature and in industry,” in *High Confidence Software Reuse in Large Systems*. Springer, 2008, pp. 100–111.

[18] B. Meyer, “Applying’design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[19] B. Rubinger and T. Bultan, “Contracting the facebook api,” *arXiv preprint arXiv:1009.3715*, 2010.

[20] J. Ullberg, U. Franke, M. Buschle, and P. Johnson, “A tool for interoperability analysis of enterprise architecture models using Pi-OCL,” in *Enterprise Interoperability IV*. Springer, 2010, pp. 81–90.

[21] O. A. Specification, “Object constraint language.”

[22] M. Buschle, P. Johnson, and K. Shahzad, “The enterprise architecture analysis tool–support for the predictive, probabilistic architecture modeling framework,” in *AMCIS 2013*, 2013, pp. 3350–3364.