Automatic Generation of Interoperability Connectors using Software Product Lines Engineering

Boubou Thiam Niang[1,2], Giacomo Kahn[1], Nawel Amokrane[2], Yacine Ouzrout[1], Hamza Sahli[2], Mustapha Derras[2] and Jannik Laval[1]

[1]*Univ. Lyon, Univ. Lyon 2, INSA Lyon, Université Claude Bernard Lyon 1, DISP-UR4570, 69676 Bron, France*

[2]*Berger-Levrault, 1 Pl. Giovanni da Verrazzano, 69009 Lyon, France*

Keywords: Code Generation, Information System, Interoperability, Model-driven Engineering, Software Product Line Engineering.

Abstract: Information Systems (ISs) of modern companies must be reactive and capable to communicate with third-party IS. It is, therefore, necessary to establish agile interoperability between information systems. For this purpose, connectors are used to ensure interactions between IS components. However, components are independently designed and conform to different technical and domain standards that are continuously evolving. Given that the connectors are often developed manually and are not reusable in most cases, establishing and maintaining interoperability becomes a major challenge. In this paper we propose an approach to automatically generate connectors, using software product line engineering to reuse common features and better manage variability. A metamodel of the connector is proposed to show that they can be considered as first-class entities. An industrial case study followed by a discussion is proposed to demonstrate the feasibility of our approach.

1 INTRODUCTION

Companies have to interact with many partners, which are structurally diverse and operate in various business sectors. In addition, companies have to evolve from a technical and organizational point of view through mergers, acquisition of subsidiaries or technological migrations. Despite their diversities, frequent changes, and independent development, the Information Systems (ISs) of different stakeholders and their components must remain compatible with each other in order to work together at all times. Connectors (Mehta et al., 2000) are used for this purpose to enable a successful interaction between component.

Some works have been carried out on the effective implementation of interoperability connectors. In (Aldrich et al., 2002; Arellanes and Lau, 2017; Garcés et al., 2019) the authors focuse on the notion of exogenous connectors *i.e.,* connectors separated from business logic components to increase decoupling and flexibility. However, this approach do not address the automation of connectors implementation.Other proposals such as (Seinturier et al., 2012; Roth et al., 2018) are interested in automatic reconfiguration for services orchestration, but the notion of connectors remains implicit. Then, some approaches have addressed the automatic synthesis of the connectors as first-class entities like (Bencomo et al., 2013; Bennaceur and Issarny, 2014; Bouloukakis et al., 2019; Autili et al., 2019). These solutions generally target a specific type of connector to be synthesized as a peer-to-peer exchange, without offering more flexibility to generate other types of connectors.

Having identified that the cited approaches do not take into account all the interesting points mentioned in the cited works, and the shortcomings identified for each of the listed works, we raised two research questions (RQ):

RQ1: To what extent it is possible to decouple the connector by separating the interaction-specific code from the business code? The objective of this RQ is to clearly identify what falls under interaction codes and and what falls under business logic in order to generate the connector.

RQ2: How to facilitate the connectors implementation despite their variability? The objective of this RQ is to show that different types of connectors can be generated regardless of its variability by minimizing the effort and reusing the connectors features.

Niang, B., Kahn, G., Amokrane, N., Ouzrout, Y., Sahli, H., Derras, M. and Laval, J.
Automatic Generation of Interoperability Connectors using Software Product Lines Engineering.
DOI: 10.5220/0011278000003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 449-456
ISBN: 978-989-758-588-3; ISSN: 2184-2833

This paper consider connectors as first class-entity and the set of connectors is considered as a Software Product Line (SPL) (Clements and Northrop, 2002) in order to exploit the commonalities of the connectors and easily manage their variability. The main contributions of this article are the following:

- Analysis of connectors commonalities and variability to consider them as a software product line,
- Design of a feature model (FM) that represents commonalities and variability of connectors,
- Proposal of a connector metamodel with reusable artifacts,
- Proposal of a software product line approach for generating interoperability connectors.

Based on an industrial case study where two connectors have to be implemented between different applications in various domains, we discussed the approach and gave feedback on the lessons learned. The metamodels, the feature models and the implementation example of the study discussed in the article are available on a public git repository. [1].

The paper is organized as follows: in Section 2, we discuss the state of the art. The proposed software product lines approach for connector generation is presented in Section 3. Section 4 presents a case study that illustrates a concrete problem to be solved. Finally, in Section 5 we discuss the approach and present feedback on the lessons learned.

2 STATE OF THE ART

Different approaches have been developed to facilitate implementation of connectors, seen in several aspects: exogenous connectors, dynamic orchestrations, and automatic synthesis.

The first aspect focuses on the separation of connectors and business logic components to increase decoupling and flexibility. Approaches such as (Aldrich et al., 2002; Arellanes and Lau, 2017; Garcés et al., 2019) go in this direction by considering connectors as first-class entities that aim to guarantee the technical, geographical, and life cycle management independence of the connectors.

The second aspect concerns the dynamic implementation of connectors via reconfiguration for component orchestration. Thus, studies such as (Seinturier et al., 2012) and (Roth et al., 2018) propose a way of composing connectors by simplifying the development of connectors. These approaches are design-time or at runtime with the possibility of reconfiguration of the interactions between the entities.

The third aspect deals with the generation of connectors. Inverardi, Tivoli and Autili (Inverardi and Tivoli, 2013; Autili et al., 2018) propose a method for automatic synthesis of modular connectors, a composition of independent primitive sub-connectors called mediators, which realize a mediation pattern. Proposition like Bencomo *et al.* (Bencomo et al., 2013) or Bennaceur and Issarny (Bennaceur and Issarny, 2014) go further by focusing on functionality discovery and automatic generation of connectors on-the-fly. To this end, both combine machine learning techniques and ontological reasoning. Bouloukakis *et al.* (Bouloukakis et al., 2019) introduce a solution for the automated synthesis of mediators that ensure the interoperability of heterogeneous things, Data eXchange (DeX) connector model. This solution aims to devise a generic connector that comprehensively abstracts and represents the semantics of the various middleware protocols. Autili *et al.* (Autili et al., 2019) define a model-based framework for mediators and an automated approach to mediator synthesis. the model takes as input an interaction process, a choreography specification and its ontology to produce what is called coordination delegate.

The state of the art show some limitations that may constraint open and evolving interoperability. Studies that address the notion of exogenous connectors do not explain how to implement them. Solutions that allow the dynamic reconfiguration of services are highly focused on orchestration where the connector is centralized. The notion of exogenous connector is not put forward for these types of approaches. It is therefore difficult to change a single connector without impacting other connector's combination. Approaches that deal with automatic connector synthesis for creating connectors are mainly focused on choreography and does not address the orchestration. Our approach aims to overcome some of the limitations mentioned in the state of the art.

3 THE CONNECTORS PRODUCT LINE

3.1 Proposed Software Product Line Engineering Approach

In accordance of the software product line engineering, the proposed approach follow two sub-process: Domain Engineering (DE) and Application Engineering (AE). Each of the sub-processes can be divided

[1] https://cvs.disp-lab.fr/demo-approach-spl-dop-connector

into two parts, the problem space and the solution space as illustrated in Fig. 1.

In both sub-processes, we rely on model-driven engineering (MDE) (Schmidt, 2006). For the DE sub-process, the architecture of the software product line is represented by the connector metamodel. In AE sub-process, the expected connector model is used for code generation, instead of directly generating the connector source code. This allows for a higher level of abstraction and flexibility. Indeed, without MDE, the SPL would have to directly implement a main class, in a target programming language, representing the connectors commonalities and variability. Thus, operations such as adding, deleting or removing can be made to this main class to vary it, and then generate the connector directly in the pre-targeted language. This kind of implementation can be achieved through delta-oriented programming with tools like DeltaJ (Koscielny et al., 2014). However, this would be a major constraint in terms of openness and scalability, as some technical choices such as the target language have to be made upstream. To avoid this, MDE provides platform independence and allows source code to be generated *a posteriori* using model-to-code transformation techniques to generate source code in a desired language.

3.2 Description of the Approach

3.2.1 Analysis of Commonalities and Variability of Connectors

The first step of the DE sub-process corresponds to the top leftmost box in Fig. 1. This step consists of identifying the commonalities and variability of the connectors. To this end, software product line engineering offers three approaches (Bakar et al., 2015): proactive, reactive, and extractive.

This work is produced with an industrial partner Berger-Levrault (BL)[2], which already operates various interoperability connectors: REST API call, Enterprise Service Bus, event-driven connectors, etc. Since we are not starting from scratch and are given the existing connectors, we opted for the extractive approach. We combine the extractive approach with the iterative approach for a variability analysis that goes beyond the existing connectors, and for future scalability of the connectors product line. The extractive approach itself can be carried out according to several extraction strategies according to Assunção *et al.* (Assunção et al., 2017). This paper relies on Static Analysis and Expert-Driven strategy extraction strategies.

Static analysis consists of analyzing the structural information of static artifacts such as source code, or textual technical and functional specification documents. To do this, we considered a repository of twenty connector projects from our industrial partner BL implementing different communication mechanisms. For most existing connectors, there is a strong coupling between the communication and the business logic source code. It was therefore necessary to first carry out a manual separation of concerns by isolating the connectors codes as much as possible. This separation of concerns addresses one of our research questions on separating interaction-specific source code from business logic (RQ1). Once this separation is made, we analyzed connectors specification documents, even for connectors that are not yet in use or have not been effectively implemented by the company. Finally, we collect additional characteristics from the literature such as the EIPs (Hohpe and Woolf, 2004) book. These three sources allowed us to define the required features for the connectors. Thus, we have identified that the connector as a first-class entity is mainly decomposed into three sub-components: source, sink, and processor which are common to all connectors. The source is the entry point of the connector which receives the message from an externe component, the sink is the output of the connector, it sends the message to a component, the processor is a sub-component which can transform the information that passes through the connector such as filtering or aggregation. Each of these sub-components can vary depending on the need for interoperability. For example, you may have a rabbitMQ or an http source, a kafka or ftp sink and an aggregator or message sequencer processor. These commonalities and variability make it easier to choose the SPL for connectors and therefore the positioning of our paper.

The Expert-Driven strategy relies on the expertise of specialists *e.g.* software engineers, architects, developers, and stakeholders, *etc*. Experts knowledge allows us to complete, adjust and validate the extracted features.

3.2.2 Modeling Connector Variability

The second step of the DE sub-process corresponds to the second box from the left at the top in Fig. 1. This step allows to represent the previously analyzed connectors features by highlighting commonalities and variability. For this purpose, we use a feature model (FM) (Kang et al., 1990), which present all possible features to create a connector. For reasons of space

[2]Berger-Levrault is a software provider specialized in the fields of education, health, sanitary, social and territorial management.

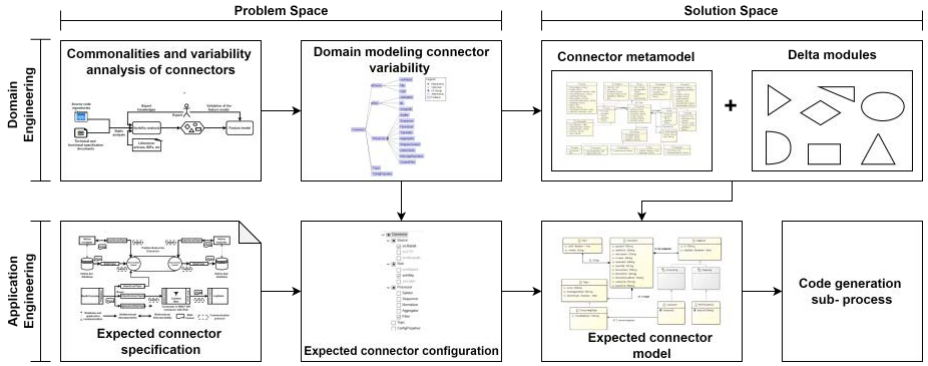

Figure 1: Overview of the software product line approach to generate connectors. The process presents the domain engineering (top) and the application engineering sub-process (bottom). For each sub-process the problem space (left) and the solution space (right) are identified.

and readability, only part of the FM is presented in Fig. 2. This later remains expandable for future evolution. A more complete FM is available on a public git repository URL indicated in the introduction.

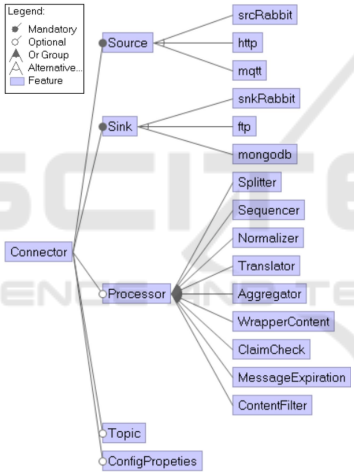

Figure 2: The domain feature model of the connectors that represents all possible features to create a connector.

3.2.3 The Connectors Software Product Line Architecture

This is the last step of the DE sub-process, the last box on top from the left. It is broken down into two phases.

- The first phase concerns the implementation of the connector software product line architecture, the connector metamodel shown in Fig. 3, which is one of the contributions of this paper. The proposal of a metamodel allows us to assert that connectors are first-class entities rather than an implicit mechanism for interaction between components. The connector metamodel is designed according to the domain FM. The transition from the domain FM to the connector metamodel is done manually.

  Indeed, the features present in the feature model become the entities of the metamodel, and different constraints and relationships must be maintained. The common features, which have a mandatory constraint, are transformed into an entity whose life cycle depends on that of the connector. This results in a composition relationship between the connector and these common entities, with the minimum cardinality equal to one. As far as the optional features are concerned, they are transformed into non-mandatory entities, which results in a composition relation with no minimal cardinality. The other constraints are represented by simple associations between entities. Features that can take a simple value becomes attributes, for example message format can be JSON or Avro. Due to space limitations and to stay readable, a portion of the metamodel is presented in Fig. 3. The complete connector metamodel is provided via public access on the provided git repository, and remains expandable to ensures evolving of the connector product line by adding new functionalities to the connector feature model and the corresponding entities in the metamodel. This extension is done manually.
- The second phase focuses on the creation of reusable operations that specify the modifications that could be applied to the connector architecture, it metamodel in this case. Indeed, these operations is due to our implementation choice which is based on Delta-Oriented Programming (DOP) (Schulze et al., 2013). The connectors product lines is represented by a core model, and a set of delta modules. The core model provides an implementation of a valid base connector. It is here about the connector metamodel.

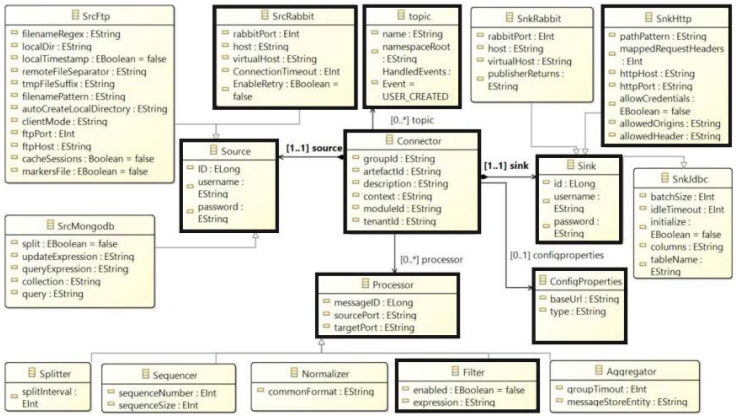

Figure 3: The metamodel of the connector representing the connectors software product line architecture. The entities with a bold border correspond to the expected connector model.

Delta modules specify the changes to be applied to the core module to create an expected connector by adding, modifying and deleting entities or their attributes. This phase consists in creating a number of delta modules, each of them is created for a valid configuration of the connector feature model. Then, for each delta module, the difference with the metamodel is evaluated that constitute so-called delta actions for later use. It should be noted that delta actions can be calculated between two delta modules, *i.e.*, one delta module is considered as a core model to create another delta module. In this case, constraints on the order of application of the module deltas are created. The industrial case study proposed in Sec. 4 will illustrate the application and use of delta actions through a concrete case. To create delta modules and delta actions, we use SiPL (Pietsch et al., 2015), a model-driven delta-oriented tool offered on top of the Eclipse modeling framework (EMF) (Steinberg et al., 2008).

3.2.4 Application Engineering

The AE sub-process leverages the domain engineering one to create a connector. It consists in four steps. The first step, the leftmost box at the bottom in Fig. 1 is to specify the desired connector. In the second step, a stakeholder configures the domain feature model by selecting the required feature for the connector specification. The third step, second to last box from the left, is the synthesis of the expected connector model. This is done on the basis of the connector product line architecture, delta modules and constraints created in the problem space of the DE sub-process, and the desired connector configuration based on the domain feature model. Delta actions are created with different constraints in the DE sub-process. Each delta action is created for a given configuration. Thus, in the current step, all delta actions corresponding to the configuration of the feature model are automatically proposed, while respecting constraints such as the order of application. The application of these delta actions allows to automatically obtain the expected connector model. The connector model is then instantiated to generate the source code of the connector in the desired language. The last step, last box from the left at the bottom in Fig. 1 is a sub-process that involves the use of certain Eclipse-based tools. To transform the model into code, the Eclipse GenModel tool is used to generate the model code which then instantiates the expected connector with property values provided by a end user.

An example of an application showing concretely how each step of the entire process works is presented in the section. 4.

4 INDUSTRIAL CASE STUDY

4.1 Scenario Presentation

To illustrate our approach, we consider a concrete case study from our industrial partner BL as shown in Fig. 4. It concerns a part of an information system where some internal components have to interact: SaaS-Console, Legibase, and Helios Analyzer and Helios Scraper. For this purpose, two variable connectors are needed to answer the research question (RQ2).

1. The SaaS-Console is an administration console that allows the clients to autonomously manage their accounts and access rights related to the Software as a Service (SaaS) they use.
2. Legibase is an application that manages practical fact sheets, articles on the latest legislation and

case law developments in several areas. Legibase is used by professionals who need access to the latest versions of often evolving legislation.

3. The Helios analyzer gathers data from different information sources such as social network, scientific and political report and integrate these information in the Helios scrapper database. Helios Analyzer allows to make a weather forecast of the latest news.

Fig. 4 shows two connectors that allow the described application to interact: an event-driven connector based on publish-subscribe (Tarkoma, 2012) communication pattern, and an hybrid connector that combine one event-driven consumer and one synchronous http endpoint.

The objectives of the first connector are to synchronize user access between the SaaS-Console, the Helios scraper, and the Helios analyzer applications, and to synchronize data between the Helios scraper database and the Helios analyzer database. This connector accomplish event-driven communication using the publish-subscribe pattern. We identify interactions performed by the same connector: SaaS-Console $\leftrightarrow$ Helios scrapper, SaaS-Console $\leftrightarrow$ Helios analyzer, and Helios scrapper database $\rightarrow$ Helios analyzer database. The directions of the arrows indicate in which direction the exchanges are made.

From SaaS-Console to Helios Scraper and Helios Analyzer and conversely, when the status of a user account changes in the UserAccessTopic in which the SaaS-Console has beforehand pushed. An event is then sent to Helios Scraper and Helios Analyzer through the connector. Events are then delivered to consumers subscribed to the topic, in this case Helios scraper and Helios analyzer. In this study case, connectors source consume the event from SaaS-Console. Upon receiving the event, components subscribed to the topic can consume the message from the sink of the connector. This is partly the same as for the flow SaaS-Console to Legibase. The difference is that for the communication from the connector to Legibase, the connector sink calls the API exposed by Legibase, upon receiving the event. The flow from Helios scraper and Helios analyzer operates in the same way, with the difference that the communication is between databases and not between business applications.

A first effort has been made by our industrial partner BL with the creation of BL-MOM, a messaging-based API (Application Platform Interface) that aims to support implementing a new connector according to the publish-subscribe communication pattern and based on the RabbitMQ broker. However, to date, BL-MOM only supports the development of asynchronous connectors. Another problem with BL-MOM is that backwards compatibility is not obvious. Indeed, our industry partner BL has found that adapting the BL-MOM API to create connectors based on a Kafka broker, for example, instead of a RabbitMQ broker requires an important development cost. In this specific case, our approach overcomes these limitations in terms of backwards compatibility, connector type and development time.

The present exchange flow case study is about a new need to connect the SaaS-Console to Legibase for access contract management, connect the SaaS-Console to Helios Scrapper and Helios analyzer for user access creation and update purpose, and connecting Helios Scrapper and Helios analyzer database for synchronization.

4.2 Implementation of the Use Case

we consider one of the connectors presented in Fig. 4 to prove the feasibility of our approach, the connector that enable communication between SaaS-Console and Legibase components.

1. We need first the specification of the connectors. For this, we consider the connector that enable interoperability between SaaS-Console and Legibase. The objective is to generate a connector that receives an event concerning user access to which it is subscribed via a topic and to make a HTTP call to the Legibase component. It is possible for the connector to filter some information thanks to the filter processor as shown in Fig. 4
2. Once the connector specified, the domain feature model must be configured by selecting the required features. The configuration is realized using SiPL tool as explained in Section 3 and Fig. 1.
3. As reminder, in the DE sub-process shown in Sec. 1, several delta modules have been created for different configurations. For each delta module the difference with the connector metamodel is calculated to obtain the delta actions, that must be applied to obtain the model of the expected connector. The resulting connector model is shown in Fig. 3, when only the entities with the bold border are considered. In other words, the model of the expected connector corresponds to the metamodel of the connector presented in Fig. 3, from which several entities have been removed such as: ftp sources, rabbit sink, and all processors except filter.
4. The last step focuses on the sub-process of transforming the resulting model of the expected connector into code. These codes are generated us-

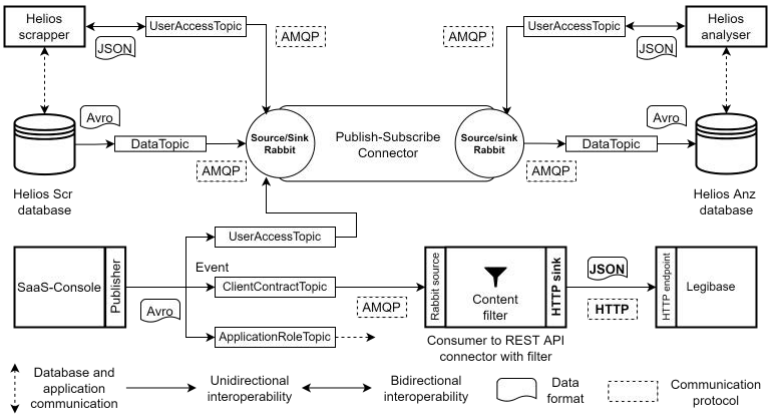

Figure 4: Overview of the industrial use case showing several components that must interact through two variable connectors based on interoperability requirements.

ing Eclipse GenModel tool. The generated code represent an API that we can use to manipulate the model. Next, we create an Eclipse fragment project to instantiate the model with the required values for the connector, using the API generated from the expected connector model , and based on EarlyBinding techniques. As a result, we get an instance of connector in the XMI format. To move on to the automatic generation of the connector code itself, we created a code generator module with our industrial partner BL. This module requires a JSON file as input and generate a connector code in java. Simply convert the XMI file to JSON and run the automatic code generation add-on.

An example of an automatic end-to-end connector generation project based on the proposed approach is available on the indicated gitlab repository. The project contains four sub-projects: feature modeling, ecore modeling, delta modeling with SiPL, and eclipse fragment project.

5 DISCUSSION AND LESSONS LEARNED

The proposed approach helps to reduce the effort required for the development of connectors by reusing their commonalities.

We made sure that the analysis of variability was as complete as possible. For this purpose, we considered concrete connectors used in the industry, or open source tools such as Spring Cloud Data Flow (SCDF) [3], and Kafka Connect [4]. We also retrieved features found in the literature, such as from the EIPs. This rigorous analysis of commonalities and variability has provided us a feature model that cover a wide range of connector types. The paper also explains the transition from a feature model to a software product line architecture manually and following rules we described. We have also shown how to use the feature model and the product line architecture to easily create a desired connector. For this purpose, delta-oriented programming was exploited using the SiPL tool and other Eclipse tools such as GenModel.

The DE sub-process being set up only once with punctual evolution, by considering the steps of AE only, we noted with the presented example that the generation of source code process based on our approach take less than one hour. This is very encouraging in terms of time saving, as according to feedback from BL developers and experts, it takes about a week for a novice to develop a connector for a single exchange flow. Although feedback from our industrial partner, BL confirms that an experienced developer takes less time, about three hours, it is always advantageous to start with an existing base project, this reduces, for example, the number of copy/paste operations that need to be performed without forgetting anything. Moreover, the fact of having a common core codes makes it possible to reduce the number of tests to be carried out, in fact the base of the product is tested only once.

6 CONCLUSION

We have adopted model-driven engineering to be platform independent, and have gone as far as possible with the code generated in Java. In the future we need to prove the code generation in other languages. We will also need to be able to manage the life cycle of

[3]https://spring.io/projects/spring-cloud-dataflow

[4]https://docs.confluent.io/platform/current/connect/index.html

connectors. Indeed, once the connector is generated, we must be able to make it evolve without having to regenerate all its source code. Considering life cycle management, the proposed approach can also be used for low-coding purposes in component-based connectors such as the SCDF connectors. In this case, we have a catalog of components sources, sinks, and processors to create connectors. The approach could thus be used to generate configurations allowing the creation of complex connectors through the composition of various sub-components.

REFERENCES

Aldrich, J., Chambers, C., and Notkin, D. (2002). Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 187–197. IEEE.

Arellanes, D. and Lau, K.-K. (2017). Exogenous connectors for hierarchical service composition. *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 125–132.

Assunção, W. K., Lopez-Herrejon, R. E., Linsbauer, L., Vergilio, S. R., and Egyed, A. (2017). Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, 22(6):2972–3016.

Autili, M., Inverardi, P., Spalazzese, R., Tivoli, M., and Mignosi, F. (2019). Automated synthesis of application-layer connectors from automata-based specifications. *Journal of Computer and System Sciences*, 104:17–40.

Autili, M., Inverardi, P., and Tivoli, M. (2018). Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. *Science of Computer Programming*, 160:3–29.

Bakar, N. H., Kasirun, Z. M., and Salleh, N. (2015). Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. *Journal of Systems and Software*, 106:132–149.

Bencomo, N., Bennaceur, A., Grace, P., Blair, G., and Issarny, V. (2013). The role of models@ run. time in supporting on-the-fly interoperability. *Computing*, 95(3):167–190.

Bennaceur, A. and Issarny, V. (2014). Automated synthesis of mediators to support component interoperability. *IEEE Transactions on Software Engineering*, 41(3):221–240.

Bouloukakis, G., Georgantas, N., Ntumba, P., and Issarny, V. (2019). Automated synthesis of mediators for middleware-layer protocol interoperability in the iot. *Future Generation Computer Systems*, 101:1271–1294.

Clements, P. C. and Northrop, L. M. (2002). Salion, inc.: A software product line case study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Garcés, L., Oquendo, F., and Nakagawa, E. Y. (2019). Software mediators as first-class entities of systems-of-systems software architectures. *Journal of the Brazilian Computer Society*, 25(1):1–23.

Hohpe, G. and Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Inverardi, P. and Tivoli, M. (2013). Automatic synthesis of modular connectors via composition of protocol mediation patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 3–12. IEEE.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., and Damiani, F. (2014). Deltaj 1.5: delta-oriented programming for java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 63–74.

Mehta, N. R., Medvidovic, N., and Phadke, S. (2000). Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187.

Pietsch, C., Kehrer, T., Kelter, U., Reuling, D., and Ohrndorf, M. (2015). Sipl–a delta-based modeling framework for software product line engineering. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 852–857. IEEE.

Roth, F. M., Becker, C., Vega, G., and Lalanda, P. (2018). Xware—a customizable interoperability framework for pervasive computing systems. *Pervasive and mobile computing*, 47:13–30.

Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25.

Schulze, S., Richers, O., and Schaefer, I. (2013). Refactoring delta-oriented software product lines. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pages 73–84.

Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., and Stefani, J.-B. (2012). A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 42(5):559–583.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

Tarkoma, S. (2012). *Publish/subscribe systems: design and principles*. John Wiley & Sons.