Computer Science and Information Systems 22(1):279–310 https://doi.org/10.2298/CSIS240701009D

Mapping-Based Approach to Integration of Technical Spaces

Vladimir Dimitrieski[1], Slavica Kordic[1], Sonja Ristic[1], Heiko Kern[2], and Ivan Lukovic[3]

[1] Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
{dimitrieski, slavica, sdristic}@uns.ac.rs
[2] Institute for Applied Computer Science, Goerdelerring 9, 04109 Leipzig, Germany
kern@infai.org
[3] Faculty of Organizational Sciences, Jove Ilića 154, 11000 Belgrade, Serbia
ivan.lukovic@fon.bg.ac.rs

Abstract. In the contemporary business landscape, the seamless integration of software components and systems is vital for ensuring the unimpeded flow of information, which is key to achieving success in the market. While addressing integration issues using standardized communication interfaces is generally preferred, standards are often adjusted or disregarded due to business goals or market strategies. Consequently, developers often resort to the manual development of integration adapters. This process is time-consuming, error-prone and persists as a significant cost factor. In this paper, we address integration issues and introduce a novel mapping-based approach for structured, automated, and reusable integration. We present an automated development process for the integration adapters at a higher level of abstraction, based on model-driven software development principles. We also present a tool called AnyMap and a visual domain-specific modeling language for specifying mappings and generating adapters, and we demonstrate the approach in a practical use case.

Keywords: mapping-based integration, technical spaces, model transformations, model-driven software development, Industry 4.0.

1. Introduction

The modern business landscape is characterized by its open, dynamic, and almost limitless nature, presenting many new challenges. Key features of this landscape include globalization, intensified competition, technological innovation, rapid development cycles, flexibility, resource efficiency, decentralized production, and individualized demand. Companies are not only required to be the first to market their own products but also to offer a high degree of customization to meet the unique needs of individual buyers. Consequently, companies are rapidly adapting and engaging in collaborative processes, which necessitates the integration of their hardware and software [1–4].

Integrated systems enhance competitive advantage by providing unified and efficient access to information [5], leading to the necessity of integrating information systems (IS). The decentralization of production has given rise to new forms of partnership,

allowing companies visibility into their respective business partners' operations. For instance, in a factory with integrated systems, partners may gain insight into the production status of a specific product. This shift has also fostered new client relationships, enabling customers to monitor the progress of their products from manufacturing to delivery. Continuous data exchange is essential for implementing standardized business processes as part of integration.

One defining aspect of modern businesses is the emphasis on technological innovation, encompassing the Internet of Things (IoT) with its Radio-Frequency IDentification (RFID) tags, sensors, actuators, and mobile phones, Cyber-Physical Systems (CPS), Internet of Services (IoS), and Smart Factory. These advancements have given rise to the concept of Industry 4.0. “Industry 4.0 is a collective term for technologies and concepts of value chain organization. Within the modular structured Smart Factories of Industry 4.0, CPSs monitor physical processes, create a virtual copy of the physical world, and make decentralized decisions. Over the IoT, CPSs communicate and cooperate with each other and humans in real time. Via IoS, internal and cross-organizational services are offered and utilized by value chain participants” [6]. Introducing protocols and data formats for device communication by market entrants can create integration challenges.

An entire company's supply chain, encompassing levels such as material procurement, manufacturing, storage, transportation, and sales, is orchestrated through an information system (IS). Each level requires data from the level below to provide services and information to end users, making integration between levels a critical concern. Therefore, seamless information flow must be facilitated through device and IS integration to ensure the company's smooth operation across all levels. In general, integration in system and software development can be defined as “the process of linking separate computing systems into a whole so that these elements can work together effectively” [7]. Developers of ISs face the challenge of connecting the machines at a software level with the existing machine landscapes and IoT.

In contemporary manufacturing systems, integrating different devices and software components can be a complex task due to the use of various data formats and communication protocols. Manufacturers often opt for proprietary protocols over standardization for various business and market reasons, which complicates integration. As a result, integration engineers are required to create adapters to facilitate smooth communication and data exchange between these devices and software components. Data formats, which are either implicitly defined or explicitly expressed by data schemas, and the tools used to handle them constitute a Technical Space (TS). Adapting data from one TS to another, such as from Comma Separated Values (CSV) to Extensible Markup Language (XML), requires the development of specific adapters for each combination of TSs. This challenge, termed inter-space heterogeneity or data model heterogeneity [8], persists despite industry standards and remains a significant cost factor [9]. Manual development of adapters for each pair of technical spaces is time-consuming, error-prone, expensive, and labor-intensive.

Despite numerous approaches and tools to solving problems in the data integration domain, there remains a significant space for improvement. Most existing tools provide just one visual concrete syntax, graphical or tabular, with no exposed or explicitly defined abstract syntax. The graphical syntax is suitable for integration scenarios with low to medium complexity, while tabular syntax is suitable for scenarios with high complexity. Having multiple syntaxes could provide different views on top of the same

abstract syntax. Thus, the tool can be used in a variety of scenarios, enabling automated model validation and reasoning and easier development of code generators. Another important issue is the granularity level of reusable components. Many integration tools provided the reuse of one of the following concepts: user-defined functions, TS specifications, or mapping specifications. The most important type of reuse is the mapping specification reuse, but it is the hardest to accomplish. With this type of reuse, whole mappings could be reused and adapted to a new integration scenario, resulting in the (semi-)automation of the integration process. This requires the reuse and adaptation at the level of individual element mappings, not just at the level of whole mapping, as is the case with most tools.

Our goal is to address the inefficiency when creating integration adapters by introducing an innovative approach for the structured, automated, and reusable integration of different TSs. In this paper, we propose an automated development process for software integration adapters at a higher level of abstraction, rather than manually crafting adapters at the data level. In our approach, we distinguish between the transformation logic and its implementation. This differentiation is facilitated by leveraging Model-Driven Software Development (MDSD) principles. As stated in [10], MDSD is a "methodology for applying the advantages of modeling to software engineering activities". It is grounded in explicitly specifying models considered first-class artifacts of all software engineering activities. Consequently, any software-related artifact is viewed as a model or a component of a larger model. The encouragement examples of successful MDSD principles' application in industry are encountered: in the development and management of Cyber–Physical Systems (CPSs) and smart manufacturing [11], in the fields of Internet of Things (IoT), and multi-agent systems [12] and in the development of software for robots [13].

MDSD approaches are usually centered around a language that is specific to a certain domain of application [14, 15]. Such languages are called Domain-Specific Modeling Language (DSML). We have developed a visual DSML aimed at specifying transformation models, known as mappings, between diverse technical spaces and their schemas. This language allows expressing of expert knowledge with a set of domain-specific concepts and appropriate models to facilitate the creation of high-abstraction-level mappings between the elements of the integrated technical spaces. These mappings are considered the fundamental units of integration and can be conveniently reused when building new transformation models. Segregating the transformation logic by creating platform-independent mappings enables transformation knowledge to be translated to different scenarios and integration platforms. We have developed two algorithms aimed at automation of transformation model's specification – a reuse algorithm and an alignment algorithm. The reuse algorithm identifies reuse candidates amongst previously specified and stored mappings by detecting similarities between the new schema elements and the elements within the existing mappings. The alignment algorithm compares pairs of source and target schemas to estimate their similarity and proposes the best matching pairs. We have developed various comparators, which can be combined to enhance the algorithms' precision. Output for both algorithms is a set of mappings that can be automatically applied to the new schema.

The executable transformation code, an integration adapter, is generated automatically from mappings for supported execution environments. This automation minimizes development effort and increases the quality of adapters.

To implement our approach, we have developed a tool called AnyMap [16, 17] for specifying mappings and generating adapters. This tool provides the following functionalities: (i) importing existing technical space data schemas or automatically deducing a simplified schema from a schema-less data file, (ii) creating mappings between the schema elements based on a DSML, (iii) reusing existing mappings based on those stored in a reuse repository, and (iv) generating executable adapters to transform the data. Adapters are generated for different target environments and programming languages, so they can be used immediately after generation on the provided source data files. Currently, a code generator is provided for our custom-built, Java-based microservice execution environment.

Therefore, our research's main contributions include our integration approach, the AnyMap tool with the developed DSML, and the implemented reuse mechanism. They are intended for software engineers of various profiles and domain experts participating in integration processes. We offer a highly generic solution that allows that concept to be applied in any business, economic, or problem domain. Following the MDSD principles and providing a formal abstract syntax for the DSML, we can specify more than one concrete syntax and automate model validation and code generation. In addition, this solution enables the reuse and adaptation of individual element mappings, not limited to whole mappings, leading to a substantial increase in the level of mapping specification automation. Finally, the automatic generation of integration adapters speeds up work, reduces the number of errors, increases quality, and reduces costs of adapter development.

The paper is structured as follows. In Section 2, we discuss related work. Section 3 presents our integration approach in detail with the developed DSML and reuse mechanisms. Afterward, we briefly present the AnyMap tool in Section 4. After that, we illustrate a use case in Section 5 and conclude this paper in Section 6 with a summary and suggestions for future work.

2. Related Work

Various methods exist for integrating TSs or system components. We focus on transferring data between interfaces or components with disparate data structures. This form of integration is commonly referred to as interconnectivity. The interconnectivity integration approaches mentioned in the literature can be categorized into two main types: standardization and transformation-based. Standardization approaches aim to provide standard solutions, protocols, and processes for different layers of the integration process. In cases where a standard is not available, fully developed, or not adhered to by a company, a proprietary protocol and integration adapters may be used to integrate desired TSs. These integration adapters are created by following a transformation-based approach, transforming input data to target data based on a set of transformation rules. This section focuses on transformation-based approaches, as our approach can also be classified under this category.

There are three main sub-categories of transformation-based approaches found in the literature: (i) schema-based integration approaches, (ii) model-driven integration approaches, and (iii) ontology-based integration approaches. What is known as schema mapping or schema matching in the database and artificial intelligence domains, in the

semantic web community is known under the name ontology alignment or ontology matching. Some of the approaches from Sections 2.2 and 2.3 also share this view and are based on ontologies, so no clear line separates these approaches and fits them into a single, separate category. Therefore, in the rest of the section, we will only focus on the schema-based and model-driven approaches.

Schema matching, as defined in the book [18], involves finding semantic correspondences between elements of two schemas. On the other hand, schema mapping, as described by Ten Cate et al. [19], is a high-level, declarative specification of the relationship between the source schema and the target schema. Visual notation is commonly used to specify schema mappings, allowing for manual specification, and may also include schema matching modules to aid in finding suitable mapping candidates. Therefore, schema matching focuses on (semi-)automatically providing a set of mapping elements, while schema mapping involves a tool for manually specifying mappings between source and target schemas, which serves as input for executable code generators.

Agreste et al. [20] provided a survey on XML schema matching. The authors expanded the scope of existing surveys on general matching approaches by describing new techniques tailored specifically for the XML domain. They argued that for the best matching technique in the XML domain, the matching tools should be specialized for that domain and utilize all its peculiarities. This approach leads to more efficient matches better suited to the XML domain, allowing for more precise identification of schema element semantics.

In their work, Bernstein et al. [21, 22] present a solution for adapting the schema mapping technique to an industrial setting. They introduce a prototype of a customizable schema matcher called PROTOtype PLAtform for Schema Matching (PROTOPLASM). This tool consists of three layers: (i) an import layer where mapped artifacts are transformed into a common internal representation based on XML, (ii) an operation layer containing the necessary concepts to construct a schema-matching strategy, and (iii) a graphical language layer used to combine graphical representations of operational concepts into matching strategy scripts for execution. Similarly, Raghavan et al. [23] propose SchemaMapper, which utilizes a hyperbolic tree instead of a linear tree representation. According to their findings, the hyperbolic tree supports faster human-performed searches for elements required during the matching process.

In their work, Alexe et al. [24, 25] present a schema mapping approach for integrating relational database schemas. Unlike traditional solutions that involve loading entire source and target schemas and creating high-level mappings between them, Alexe's "divide-design-merge" approach advocates splitting source and target schemas into smaller parts, establishing mappings between these parts and then merging all partial mappings into a comprehensive whole as the final step. The approach is accompanied by three tools developed by the authors.

Duchateau and Bellahsene introduce Yet Another Matcher (YAM) [26], a self-tuning, machine-learning-based, and extensible matcher factory tool in their work. YAM generates a best-fit schema-matching algorithm tailored to a specific integration scenario. The generated algorithm identifies schema element matches and proposes them to the user. The self-tuning feature allows the production of a matcher with user-defined characteristics for a given scenario, while the extensible feature enables users to add new similarity measures, enhancing the system's overall effectiveness. Similar

techniques are found in MatchPlanner [27], utilizing decision tree methods, and eTuner [28], which employs synthetic matching scenarios to create matches.

Model-driven software development facilitates the development of software systems at different levels of abstraction, with DSMLs playing a crucial role in reducing development costs. In MDSD, transformations are defined at the meta-model level, wherein data schema and transformation rules can be interpreted as schema-matching rules. Büttner et al. [29] have introduced a model-driven approach for integrating data among government institutions in Germany. The approach emphasizes standardizing the messages, interfaces, and data models exchanged. Adherence to these standards is overseen by a central governing body that defines meta-models (data formats) for various sectors within the German government. Given different standards, integrating data becomes a crucial task for enabling exchange. This integration process operates at the meta-model level, allowing for the transformation of messages and facilitating communication with other German or European institutions.

Authors of [30, 31] have introduced a meta-modeling approach for integrating heterogeneous distributed IT systems, known as Berlin Brandenburg Business Process Integration and Evolution framework BIZYCLE. The BIZYCLE integration process relies on multilevel modeling abstractions. Initially, the integration scenario is modeled at the computation-independent level, covering the business aspects, and then refined at the platform-specific level to describe the technical interfaces of the integrated systems. A platform-specific model is created for each supported platform. The integration process is automated through model extraction, systematic conflict analysis, and code generation. The BIZYCLE Repository [32] supports reuse at the model level, allowing interface descriptions, transformation rules, and semantic annotations to be stored and shared between projects and users.

There are various DSMLs and frameworks that are not directly linked to schema mapping but are better suited to the fields of schema matching and enterprise application integration. Vuković et al. [33, 34] have introduced a language called Semantic-Aided Integration Language (SAIL). This language allows the description, generation, and use of matching components within their framework without requiring implementation in a general-purpose programming language. The developed matching framework aims to automate certain steps in conflict resolution during the matching process. Interfaces and their elements can be semantically described using ontologies to facilitate this automation. Although the approach is based on ontology alignment principles, the SAIL domain-specific language is used to specify matching algorithms and adheres to the principles of the MDSD methodology. Similarly, with the IS modeling approach based on MDSD principles and the form type concept, Luković et al. [35, 36] propose integrated modeling of disparate parts of an IS.

The FUSE (Federated User Exchange) approach [37] is a domain-aware method for achieving user model interoperability. It involves manual mapping and automatic translation processes, each utilizing two domain-aware mechanisms: (i) a canonical user model and (ii) user model mapping transformations tailored to specific domains. All mappings are initially made with the canonical user model as the target, which serves as a consistent shared user model. The user model mapping transformations are components designed specifically for mapping between different user models via the canonical model. This approach differentiates itself from generic approaches by integrating domain knowledge into new processes and tools, supporting complex user model interoperability tasks across overlapping domains.

In summary, despite the availability of numerous schema-matching approaches, only a small subset of them is actively being developed and maintained. Most accompanying tools were created as prototypes to validate the approach, making them outdated or inapplicable for real-world scenarios. Also, these approaches focus on schema-matching in relational databases and XML domains, traditionally seen as the training ground for schema-based integration algorithms. However, only a few identified approaches have been applied outside these domains, with the most notable instance mentioned in [12] being in the industrial domain. In contrast to these approaches, which mainly focus on matching source and target elements, our approach takes a broader view of the integration problem. It encompasses all the steps preceding the specification of rules and incorporates both manual and automatic integration mechanisms. As observed in the surveyed approaches, our tool can leverage schema and ontology-matching algorithms to facilitate process automation.

Some of the fundamental elements of our approach, such as the previous iterations of DSML, our integration tool, execution environment, and the reuse algorithm, have been outlined throughout our previous work [16, 17, 38, 39]. However, this paper marks the first time we have formulated and presented our integration approach in its entirety, also encompassing all the updated details since the previous publications. The meta-model of our DSML has been enriched with additional concepts to offer broader domain coverage. Based on the results of applying the language in several use cases, existing concepts have been also fine-tuned to better fit the domain. In addition, the automation feature now incorporates an alignment algorithm that does not rely on previously created mappings to streamline developer efforts. This notably enhances the level of automation in cases where the reuse repository has not yet been populated with previously created mappings, thereby rendering the reuse algorithm inapplicable. Lastly, we present a realistic, more intricate industrial integration example covering all aspects of this improved approach.

3. Mapping-based integration approach

In this section, we present our mapping-based integration approach, which is the main aspect of our research. The approach focuses on enabling interconnectivity, which maintains the existing system functionality while enabling data-level integration. It employs a common data structure onto which all other data formats are mapped to facilitate the integration. This common data structure can be seen as a TS model. Adhering to the MDSD principles, each model is treated as a first-class citizen, and all necessary operations are specified and performed on top of them. These TS models are directly utilized in the development process of integration adapters by establishing transformations between source and target TS model elements at a higher level of abstraction. We denote these transformations as mappings, and they are created utilizing a visual DSML. Once all the relevant mappings are specified, based on these abstract yet formal specifications of integration adapters, code generators can be used to generate executable integration adapters.

While the entire integration adapter can be manually developed in a chosen programming language, this process is usually overly repetitive, time-consuming, and error prone. Offering an integration tool to support the adapter development increases

the level of automation in this process, yielding better development times and fewer errors. Therefore, the description of the approach in this section assumes that the integration with our approach is carried out with the support of an integration tool.

Each activity of the integration development approach can be performed either by the adapter developer, automatically by the tool, or semi-automatically with the developer and tool participating in the activity execution. To reflect this, all diagrams depicting process steps in this section have two swimlanes. The first swimlane is for the developer, where all the integration process steps are performed by the developer. The second swimlane is for the integration tool, where all the integration process steps are performed in an automated manner by the tool without any developer intervention. Any process steps that can be performed by the developer, the tool, or both are drawn at the border between the two swimlanes.

To facilitate the automated development of integration adapters, our approach can be divided into three phases:

1. *Import of TSs*: An adapter developer imports all participating TSs into an integration tool.
2. *Mapping specification*: The developer specifies the source-to-target mappings between the TSs imported in the previous phase. Specification may be completed manually, using our DSML, or in an automated way, using a supported mapping automation facility.
3. *Generation of an integration adapter*: The executable adapter code is generated for a chosen execution environment based on the mapping specification created in the previous phase.

3.1. Phase 1: Import of TSs

To enable data-level integration between two TSs, i.e., interconnectivity, developers must first provide data schemas and data files from those TSs. These files are the input to the *Import TSs* phase presented in Fig. 1.

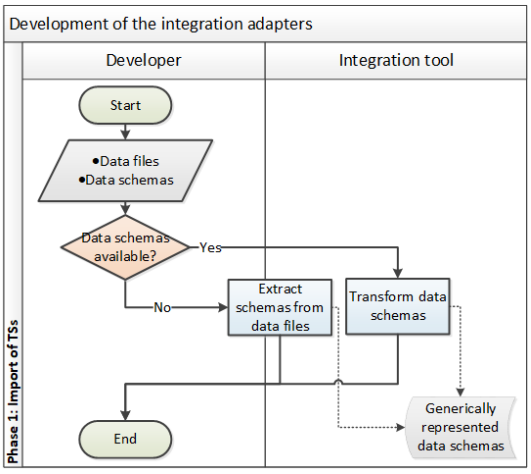

Fig. 1. Activity diagram for the *Import of TSs* phase

Data files serve a dual purpose. First, they are used during the adapter execution as they are transformed into another file corresponding to the target TSs. Second, if no data schema is available, they can be used as examples to partially or fully construct the schema. These data files are referred to as example data files.

Data schemas are essential for the integration adapter development as they define the structure for creating mappings. If the data schema exists, it is imported via the *Transform data schemas* process step. If a data schema is missing, a developer can use example data files as input for the (semi-)automatic extraction of schemas (*Extract schemas from data files*). This presents a challenging task, akin to inferring DSML context-free grammars from example models written in that DSML [40]. By having enough example data files, an inferred schema may closely match the original. Nonetheless, even a small subset of example data files can yield a simplified schema with sufficient information for integration. Therefore, for some TSs, schema extraction is a straightforward process and can be performed automatically, for some TSs, user input is required to adequately construct the schema document. Therefore, we classify this step as (semi-)automatic and put it in the middle between the two swimlanes in Fig. 1.

For the integration process to be applied consistently, imported data schemas must be represented in a common, generic manner, irrespective of the TSs being integrated and their specificness. We view each data schema as a graph of nodes and links describing their relationships, as introduced in [41] and described in short in the rest of the text. A schema can be represented as a graph with a single root element and multiple child elements. The root element can either be explicitly specified in the schema or be a part of the shared substructure and referential constraints. We have used a tree representation for all schemas to make it easier to handle and represent schemas. General graph schemas are converted into tree schemas by flattening the graph structure. During the flattening, relationships between elements at the same level or between upper and lower levels of the tree can be represented as tree elements with non-trivial types. This can be achieved by copying referenced structures to the referenced place or introducing a special reference element type for representing references where copying is not an option, as it would introduce recursive and infinite structures. Flattening is done at the TS importer level. The importer's developer is responsible for converting the original schema structure to the generic tree representation. The flattening can be implemented in many ways and algorithms, one of which is described in more detail in [41]. This generic schema representation is the output of *Extract schemas from data files* and *Transform data schemas* process steps and serves as a base structure for creating mappings. Concepts used to represent this generic schema structure are a part of the meta-model of the developed DSML and are presented in more detail in Section 3.2.

3.2. Phase 2: Mapping specification

Once the generic schema structures are created, transformations are specified as mappings between schema elements. In our approach, this is done by utilizing a custom-made DSML and an expression language, described in detail later in this subsection. To start the mapping creation, a developer performs the *Mapping specification* phase of the approach, depicted in Fig. 2.

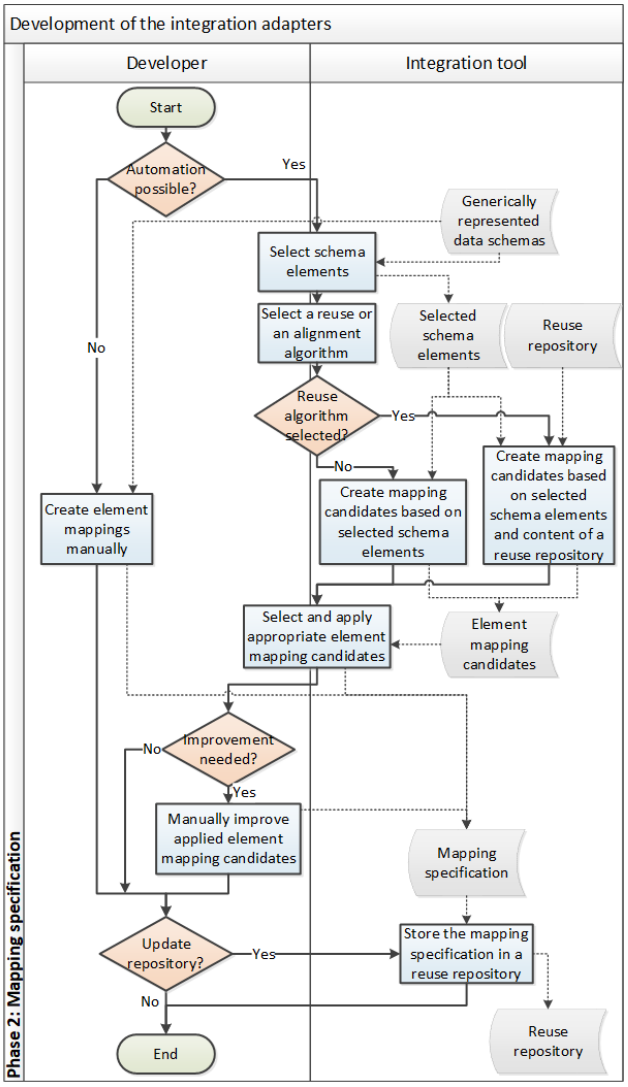

Fig. 2. Activity diagram for the *Mapping specification* phase

Although our integration approach promotes using automated facilities for mapping specification, we also acknowledge some cases where a manual specification utilizing our DSML is required. For example, one automation facility reuses previously defined mappings stored in the reuse repository and tries to apply them to the current integration scenario. Manual mapping is required when there is not enough historical data for reuse to autonomously and with high precision conclude the best possible mappings for the current pair of TSs. Therefore, this section discusses the three ways of specifying mappings between the generic schema structures: manual, semi-automatic, and fully automatic. Regardless of the chosen way, the mappings are created by utilizing the

provided DSML, and the result is a mapping specification used to generate the executable adapter code.

The manual way of creating mappings (*Create element mappings manually*) allows developers to use our DSML to manually connect the source and target schema elements and provide additional transformation expressions that will later be used at execution time. It must be noted that this is different from manual adapter development in a programming language, as in this case, the developer uses the DSML to formally create such an adapter without the integration tool's automation mechanisms in use.

Implementing automation at the schema level allows multiple data files to be easily transformed using the same mapping, making automation more efficient. However, mapping reuse is restricted as there is significant variability at the data level, even for data that conforms to the same schema element. Conversely, if a mapping is specified at the schema level, it can be easily adapted to another schema element and transform all instances of both the new schema element and the original schema element.

There are two approaches to automating the creation of mapping specifications: fully automatic and semi-automatic. In the case of a fully automatic approach, all schema elements are passed to the automation algorithm as it tries to specify all possible mappings that make sense for the two TSs in hand. As this way of mapping specification takes time to calculate the possible mappings, a developer might choose the semi-automatic method due to the usually large search space comprising many source and target schema elements. If a developer chooses this way for mapping specification, the first step is to select a subset of source and target schema elements that need to be mapped (*Select schema elements*). While this step is not always necessary, it may significantly speed up the automation process for large schemas.

Regardless of the automation method, fully or semi-automatic, a developer must choose an automation algorithm to proceed, either a reuse algorithm or an alignment algorithm (*Select reuse or an alignment algorithm*), both introduced later in this section. Executing the alignment (*Create mapping candidates based on selected schema elements*) or reuse algorithms (*Create mapping candidates based on selected schema elements and content of a reuse repository*) requires a set of source and target elements as input and produces a set of mapping candidates as output.

In the next step (*Select and apply appropriate element mapping candidates*), some or all candidates are applied to the current mapping context, becoming the final mappings from which the integration adapter is generated. In the case of the fully automatic method, the offered candidates are all applied to the current integration context, which considerably speeds up the development process and reduces errors as the developer does not need to interfere with the process. In the semi-automatic method, users may choose which mapping candidates will become new mappings applied to the mapping context. If they decide so, developers might also perform the *Manually improve applied element mapping candidates* process step to achieve higher precision in the automation process in cases when the precision of the automation algorithm is low because of the scarce mapping history it operates on. A developer creates use-case-specific element mappings missed by the automation algorithms creating more complete mapping specifications.

The result of performing the *Mapping specification* phase is a formally defined mapping specification, which is serialized and saved to storage. The mapping specification is later used in the *Generation of Integration Adapters* phase, presented in the following subsection.

The DSML for mapping specification

We developed a new DSML that is a core component of our approach. Initially, we identified the essential concepts necessary for creating mappings by thoroughly examining existing literature and numerous integration tools available at the time. As it is stated in [42], domain experts are essential for producing reliable results when constructing, making decisions, and evaluating a language. Therefore, a preliminary set of concepts was then discussed with domain experts across various integration domains, primarily focusing on industrial manufacturing where this issue is particularly significant. Fig. 3 shows its current meta-model based on previous versions presented in [16, 17, 38]. Generically represented data schemas serve as the foundation or recipe for creating integration adapters and, therefore, are part of the DSML meta-model. Concepts used to represent data schemas are represented as rectangles with gray filling. Once the data schemas have been identified and represented in a generic manner, the mapping specification process can begin. Concepts used for the element mapping specification are represented as rectangles with a white filling. In the rest of the section, the names of meta-model concepts are given in italics.

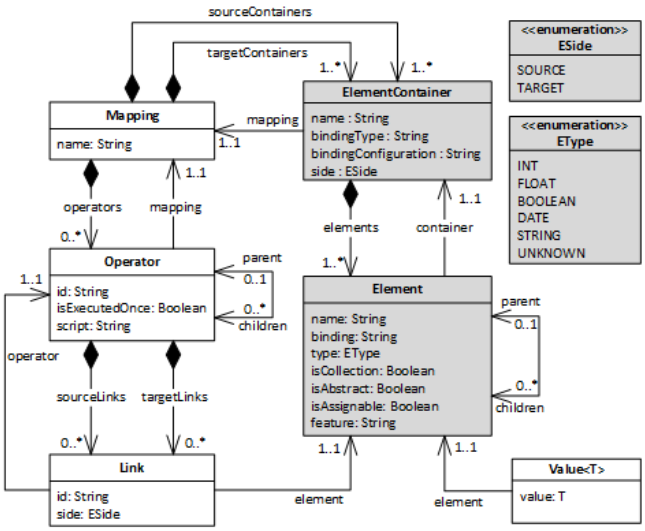

Fig. 3. The mapping DSML meta-model

A single element container (*ElementContainer*) is created when a data schema is loaded. It is given a name (*name*) to distinguish it from other containers. Depending on whether the container represents a source or a target schema, the side attribute (*side*) is set to either *SOURCE* or *TARGET*. These are both literal values defined as part of the *ESide* enumeration. Each container is contained by a single mapping specification (*mapping*), either as a source container (*sourceContainers*) or a target container (*targetContainers*).

The concept of binding refers to the link between the imported, generic schema element and the original schema element from which it originates. These bindings create a two-way connection between the original data schema and the generic schema

structure. This connection is crucial for the adapter execution process because the mapping specification process occurs on top of the generic schema representation, while the resulting adapters must be executed based on the data files with the original schema definitions. By following these bindings, the code generator can accurately produce code that accesses the original elements of the schema documents and their associated data files. These bindings are established when a TS is imported into the tool, hence why we refer to these TS importers as binders.

Therefore, at the level of an element container, a binding type (*bindingType*) and a binding configuration (*bindingConfiguration*) attribute must be set. The binding type identifies the binding component responsible for reading the data schema and handling data file read/write operations. Based on the binding type attribute, a generator can package the appropriate binder component with the integration adapter for execution. The binding configuration attribute is a storage for schema-level configuration during data schema import by the identified binder, based on the binding type. Since different binders may format this attribute differently, it's binder-specific. The binding configuration is essential for recreating the element tree during adapter execution, as the adapter only has access to the mapping file.

Each container in the system consists of one or more data schema elements (*Element*). The name (*name*) and binding string (*binding*) are an element's two most important attributes. The name of an element corresponds to the name of the corresponding imported schema element. The binding string is a binder-specific unique identifier that stores the path of the schema element in the imported data schema document. In addition to the binder-specific configuration, a binder may set an additional feature (*feature*) for the element that will be used later in the adapter execution process.

A binder can set values for each element that specify the element's type (*type*) and properties. These properties specify whether the element is a collection (*isCollection*), abstract (*isAbstract*), or assignable (*isAssignable*). By setting these values, the binder limits the number of functions that can be applied to these elements, which helps to prevent inappropriate mappings from being specified in the first place. If an element is a collection, usual collection operators can be applied. Abstract elements are placeholders or grouping elements and cannot be used as source or target elements of an element mapping. Finally, non-assignable elements can only be used as a source but not as target elements of an element mapping.

The element type attribute is only set for primitive elements, with possible values defined by the EType enumeration. Complex types, like objects, are represented as sub-trees within their element container (*children*). Each element representing an object property has a parent relationship (*parent*) pointing to the element representing the object.

The main meta-model concept used for the mapping specification is *Mapping*, which represents a single mapping specification used in the adapter generation process. Each mapping has a name (*name*) and a set of source and target element containers representing loaded data schemas. Additionally, a set of element mapping specifications (*Operator* and *Link*) specifies transformation rules at a higher abstraction level.

In addition to the language representing the connections between source and target schema elements, i.e., element mappings, it is important to have an expression language for expressing algorithms to manipulate the input values. For each operator that represents a high-level element mapping, we specify the value-level transformation

logic in the *script* property using the expression language. This expression language can be either custom-developed as a Domain-Specific Language (DSL) or adapted from a chosen general-purpose language. In our approach, the expression language is created by adapting an existing general-purpose programming language. This is achieved by specifying one or more integration-specific structures and application programming interfaces (APIs) and using the standard programming language mechanisms on top of these structures. This process is similar to building an embedded or internal DSL [43], where we use the syntax of the host language and add domain-specific elements, such as an integration-specific structure. From the point of our DSML, the transformation code written in the *script* property is related to the target framework or libraries used for the execution of the generated adapter code.

The advantage of using an existing language is that all its mechanisms, statements, and expressions are available for defining the transformation logic. To create a structure that can be used in any general-purpose programming language, we have designed a value encapsulation structure using the generic *Value<T>* concept (illustrated in Fig. 3). The template parameter *T* represents the type of the value (*value*) received as input, corresponding to the schema element (*element*). Therefore, this structure encapsulates values and schema elements, allowing expressions to be made at both meta-data and data levels. Furthermore, the *Value<T>* structure and the connected *Element* concept can be easily translated to a structure in any target scripting language, provided the language supports generics or a similar concept (e.g. if the target language is dynamically typed).

When an operator is connected through input and output links, input and output variables are automatically created in the script. These variables are instances of *Value<T>*, where *T* represents the type of the schema element connected by the link. The expression language can use these variables to specify the transformation logic. Additionally, transformation rules can be implemented in an arbitrary construct from a general-purpose language on top of these variables.

Automating the mapping specification

As introduced earlier in this subsection, users can utilize the DSML to manually specify the transformation logic or use a mapping automation facility to help them in that endeavor. The mapping automation facility is developed in response to the existence of variability in input and output data schemas. In the realms of MDSD and DSMLs, a language developer can choose to either introduce new language concepts and adjust code generators to handle newly encountered variations [42, 44] or employ automatic model refinement and creation algorithms that do not change the DSML while assisting users based on implicit or stored domain knowledge. Our approach to automatic variability handling aligns with the latter category.

There are two kinds of automation algorithms to apply to their current integration context. Fig. 4 visually represents the difference between these two algorithms. The left side of the figure depicts the alignment process. This process uses a schema-matching algorithm to identify similarities between the source schema elements ($Es_i..Es_k$) and the target schema elements ($Et_j..Et_l$). The matching algorithm compares each pair of source and target schema elements and returns a similarity score.

The reuse process is depicted on the right side of the figure. Though in a slightly different role, the matching algorithms can also be used in this process. Instead of comparing the source (Es_i...Es_k) and target (Et_j...Et_l) schema elements directly, the matching algorithms compare schema elements with the schema elements from a mapping repository, (Ers_i...Ers_k) and (Ert_j...Ert_l) respectively, to find previously created element mappings that are similar and can be reused and applied in the current integration scenario. This results in a probability that the repository mapping fits the current integration scenario. Defining a minimum probability threshold is possible to simplify choosing the appropriate repository mapping. This threshold is used to exclude any unnecessary element mappings. Throughout the rest of the text, we will simply refer to this threshold as "probability".

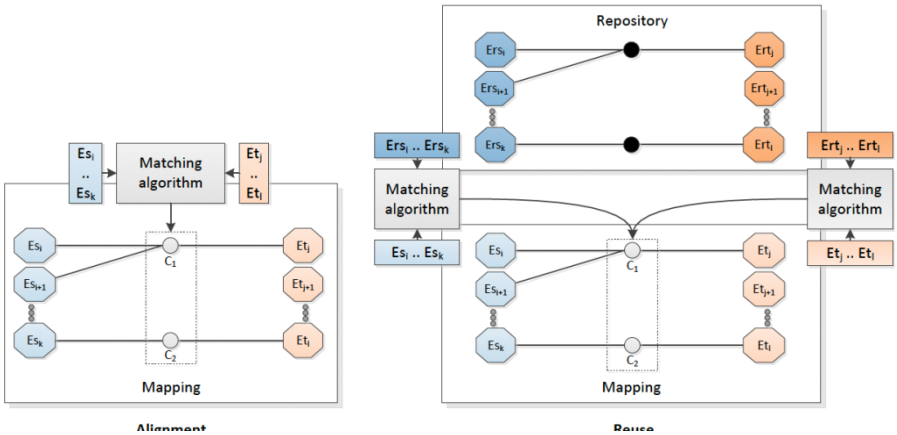

Fig. 4. Comparison of the alignment and reuse algorithms

Independent of the chosen automation algorithm, the result is always a collection of mapping candidates that can be applied to the current integration context to create mapping specifications. Such formal mapping specifications conform to the presented DSML's meta-model. As such, it possesses intrinsic value for reuse in future integration adapter developments as it is formally and generically specified. Therefore, a developer may want to improve the reuse process by adding the created mapping specification to a *Reuse repository* (cf. Fig. 2). There are two types of repositories: local and global. Local repositories are deployed on a single machine and maintained by a developer or group of developers who typically work on the same integration issues. By focusing on a single integration domain, local repositories can improve the accuracy of the reuse algorithm for that domain. However, local repositories tend to have a small number of mapping specifications, which means that the reuse algorithm cannot provide element mapping candidates for newly introduced elements with no previous similar elements in the domain. In such situations, a global repository may be more suitable. Developers from different integration domains store their mapping specifications in a global repository. Depending on the tool and data security policies, a global repository may be deployed at a company level or worldwide. More details about the automation algorithms may be found in our papers [38, 39].

3.3. Phase 3: Generation of Integration Adapters

After completing the previous process steps, a mapping specification is created. It represents an abstract specification of the integration adapter. If developers are satisfied with the mapping specification, they can generate an executable integration adapter for a specific execution platform. In Fig. 5 we present the activity diagram for the *Generation of integration adapters* phase.

Our approach envisions having multiple code generators, each responsible for generating code that runs on a different execution platform or environment. Developers can choose and invoke the most suitable code generator for their integration needs (*Select and invoke a code generator*). The integration tool executes the entire code generation process (*Generate code*) by taking a formal mapping specification as input, parsing its contents, and producing an executable integration adapter code as output. The executable integration adapter serves as the output of the entire presented integration process.

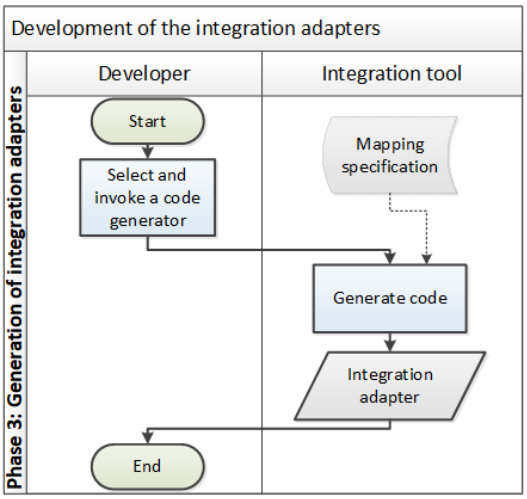

Fig. 5. Activity diagram for the *Generation of integration adapters* phase

4. The AnyMap Tool

We have developed an easily extendable integration tool named AnyMap to support our integration approach. AnyMap is the primary interaction point with the user and encompasses all activities, from reading and parsing data schemas to specifying mappings and generating an integration adapter. The tool helps us to realize and validate our approach to real-world problems. The architecture of the tool is illustrated in Fig. 6.

The AnyMap tool comprises five modules: Core, Binding, Mapping Editor, Reuse, and Generator. Within each module, one or more plug-ins adhere to the interface outlined in the core module. To add new plug-ins to a module, it is necessary to

implement the appropriate interface from the core module and register their execution with the Eclipse IDE (Integrated Development Environment) runtime engine. The Eclipse IDE is a widely used IDE for Java developers who create web and desktop applications. With Eclipse, it's easy to create plug-ins that add functionality to the IDE, allowing fast and agile development. All plug-ins are implemented in Java and Xtend programming languages.

The **Core** module is a fundamental part of the tool, including basic components used throughout the other modules. These core components have been developed to provide only the most essential functionality, allowing easy extension of other modules. The Core module comprises concepts such as the mapping language, expression language, and interfaces for implementing binders and generators. It includes all the necessary interfaces required to extend any of the other modules.

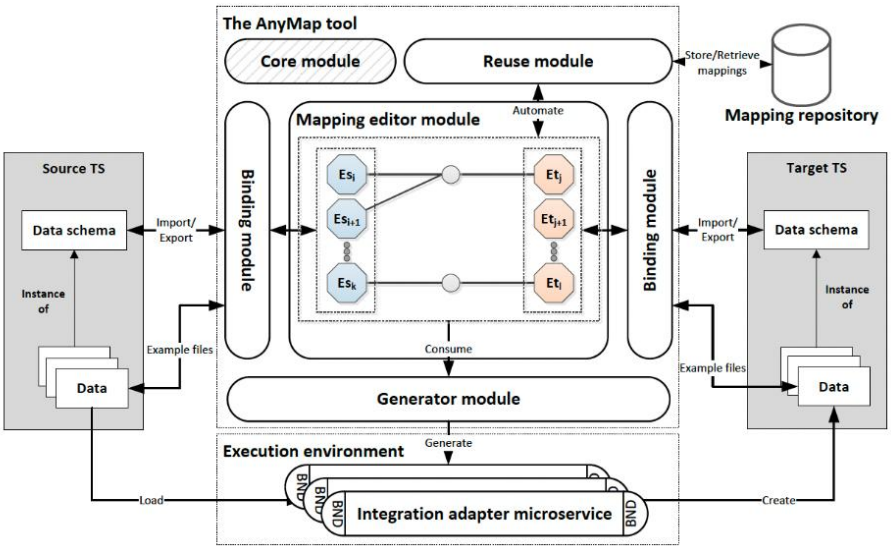

Fig. 6. Architecture of the AnyMap tool

The **Binding** module contains plug-ins representing TS binders used for importing and exporting data and data schemas of a specific TS. These binders can transform schemas into generic element trees and store reverse links from generically represented schema elements to the original schema elements. Each binder plug-in has an appropriate GUI (Graphical User Interface) that provides user interaction with the binders. For instance, when importing a new file from a TS, users can set up the binder parameters through a GUI wizard. Therefore, each binder is designed to support all the process steps presented in Fig. 1.

The **Mapping Editor** module is a crucial component in the AnyMap tool, serving as the main point of interaction between users and the system. This module offers a graphical concrete syntax for the mapping language and a textual concrete syntax for the expression language. It also provides all the necessary GUI classes, such as event listeners, commands, and menu items, to ensure seamless interaction between the user and the tool.

Currently, the Mapping Editor module implements a graphical concrete syntax of the mapping language, which we believe is the most appropriate syntax for the integration domain we are interested in. This syntax provides a comprehensive view of the mapping specification, making it easy to learn and understand. A disadvantage of the graphical concrete syntax is that the diagram becomes overcrowded when many mappings are created. However, we feel that the benefits of this syntax outweigh its drawbacks.

Our tool supports the textual concrete syntax for the expression language, as we chose Java as our expression language. We have limited the possible number of functions that can be used in creating expressions, and each expression can only be defined on top of an object that is an instance of the *Value<T>* class presented in Fig. 3.

The **Reuse** module consists of various plug-ins that work together to facilitate aligning and reusing schema elements. These plug-ins use different algorithms to calculate similarities between schema elements and generate a list of element-mapping candidates as an output. Users can then review this list and their probabilities and select the most appropriate element candidates for their specific integration scenario. Once selected, the AnyMap tool will automatically apply these chosen element mappings to the current mapping specification and store them for future reuse in the **Mapping repository**.

The **Generator** module consists of plug-ins that can be used to create executable transformations for a specific execution environment. The main function of each plug-in is to analyze a mapping file, extract the necessary mappings, and generate executable transformation code based on these mappings. Each plug-in is designed to improve the tool's capabilities in generating code for different execution environments. We have only implemented a generator for our custom-made execution environment.

We have developed a custom **execution environment** to support the execution of generated adapters in a scalable and transparent way. Although a user can provide different generators for the AnyMap tool to generate adapters for different execution environments, we have implemented an execution environment based on the microservice architecture. We chose this architecture to demonstrate that integration adapters can be generated as stateless code components and run on demand when a new file needs to be transformed. This environment has been developed in the Java programming language and using the Spring Cloud library. These components can be instantiated on-demand depending on the frequency of input data reception [38].

5. Example Usage

To illustrate the applicability of our approach, we have applied it to a typical device-to-information-system integration scenario in an industrial context. Such a scenario is frequently encountered and entails slight variations in integrated technical spaces, which is perfect for effectively demonstrating the full power of our approach. The presented use case involves integrating a sensor that measures various characteristics of semiconductor wafers and an IS module for data visualization. The integration is performed between CSV and XML TSs. The sensors gather data and send it as a CSV document. The information system visualizes the data using the JavaScript Chart (JSChart) library and expects it to be formatted according to a predefined XML schema.

The import mechanism must overcome technical (inter-space) and functional (intra-space) heterogeneity to import CSV and XML documents. Functional complexity is caused by different measurement methods that can result in variability in the structure of CSV documents, which means that an IS vendor needs a set of different adapters for integrating sensors that use different measuring methods. However, manual implementation is often time-consuming, costly, and error-prone. Our approach is used to simplify the specification of adapters in the presence of the two heterogeneity problems steps, and the tool modules used in this use case are outlined in Fig. 7.

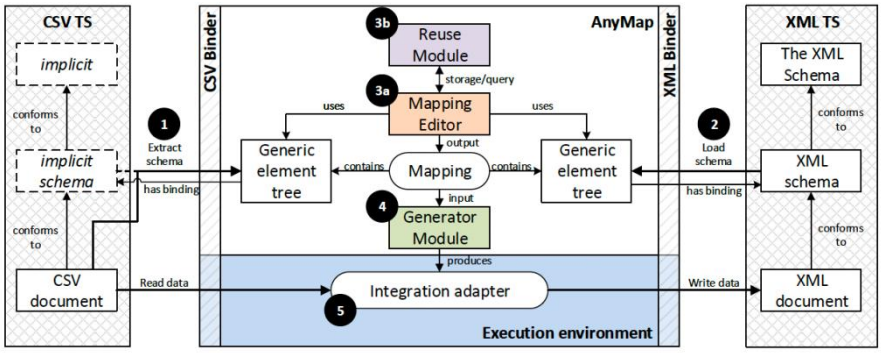

Fig. 7. Importing CSV and XML technical spaces using the AnyMap tool

Fig. 7 displays the five main steps to merge CSV and XML, numbered in black circles. The rest of this section gives a detailed explanation of the steps. In Fig. 8 we present the input CSV file and the target XML schema to which the CSV file must be transformed.

Step 1. To work with CSV files, which do not have an explicit schema definition language, the schema must be extracted from an example data file. This is done using the CSV binder to read the data file and extract schema information. The schema can be extracted by reading column names from the file header or by manually specifying column names and types.

Step 2. Creating a generic element tree from an XML schema structure is a simple way to load the schema. With the help of the XML binder, a developer can easily convert an existing XML Schema Definition (XSD) document into a generic tree representation. Unlike the CSV binder, the XSD binder is very simple. A developer only provides a path to an XSD document and the mapping side on which this TS participates.

Step 3. After importing data schemas from both technical spaces, the tool provides a blank canvas for mapping creation with a generic element tree present on both sides of the canvas. In Fig. 9 we captured a mapping state from an ongoing mapping specification based on these imported data schemas.

Meas. Nr.	Ord. Nr.	Weight	Radius	Thickness	Pos. Before	Pos. After	Fallback
1048	1	7570	7205	19.772	25.601	-0.001	0
1048	2	7279	6932	20.002	4.734	0.031	0
1048	3	7175	6837	20.248	5.528	0.112	0
1048	4	7200	6863	20.395	-5.572	-0.204	0
1048	5	7300	6957	20.262	-6.628	-0.130	0
1048	6	7555	7198	20.177	-2.361	-0.165	0
1048	7	7880	7500	19.752	100.840	0.228	0
1048	8	7669	7301	19.868	-2.013	-0.049	0
1048	9	7278	6926	19.681	-19.475	-0.175	0
...	...	...	...	...	...	...	...

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="unqualified"
    elementFormDefault="qualified">
    <xs:element name="JSChart">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="dataset">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element maxOccurs="unbounded" name="data">
                                <xs:complexType>
                                    <xs:attribute name="unit" type="xs:int"/>
                                    <xs:attribute name="value" type="xs:int"/>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                        <xs:attribute name="type" type="xs:string"/>
                        <xs:attribute name="id" type="xs:string"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Fig. 8. CSV document and XML schema examples

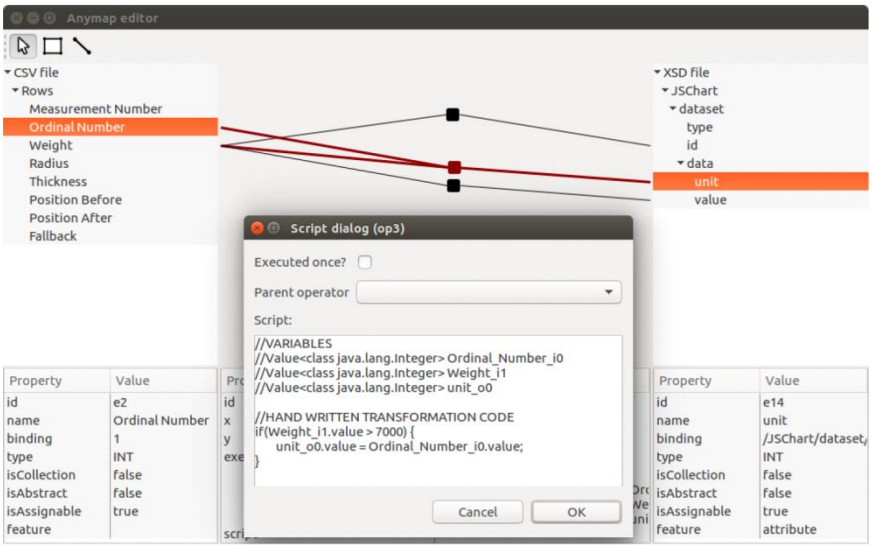

Fig. 9. The mapping specification process

On the left side of Fig. 9, there is an element container labeled "CSV file." This container holds just one element called *Rows*, which is an abstract element representing the data payload of the CSV document. By payload, we mean the rows that show measured values from the System Under Supervision (SUS). Additionally, a CSV document may have other top-level meta-attributes that don't represent measured values, but instead provide information about the protocol, sensor configuration, and manufacturer. If these elements appear in the document, they will be created on the same level as the *Rows* element. Each child element of *Rows* represents a single column from the CSV document, and these elements are not abstract. They can be used in the mapping specification, and their type is inferred during the binding process. Their binding, or reverse link to the original schema element, is an ordinal number of the column they represent. You can see these properties in Fig. 9 in the property view located below the generic element tree. The property views displayed show the properties of the Ordinal Number and Unit elements.

On the right side of Fig. 9, there is an element container that is based on the XSD document that has been imported. This element container is named "XSD file". The only child element of this container is the "JSChart" element. The "JSChart" element is created from the "JSChart" root element of the XSD document in Fig. 8. All other child elements are created from the XSD sub-elements and their attributes. The binding values are XML Path Language (XPath) expressions uniquely identifying each XSD document element.

To create element mappings, the developer must first create the generic element representations for both the source and target meta-models. Element mappings are comprised of two components - operators and links, which are specified separately. Operators are represented as rectangles and must be created first. They are then linked to generic tree elements via links, which are represented as lines. Every link connected to an operator introduces a new variable that can be used in the operator script when writing expressions using the Expression language. The variable name is derived from the element name by adding a single character that represents the side of the link in comparison to the operator (i.e. "i" for input and "o" for output) and an ordinal number of the link at its side of the operator. Examples of link names can be seen in Fig. 9 in the opened *Script* dialog.

The first few lines of the script are comments with variable names which provide a good operator overview to developers. For example, the highlighted rule which is marked red on the canvas has two inputs and one output link and therefore has three variables: (i) *Ordinal_Number_i0*, first input variable corresponding to the Ordinal Number element, (ii) *Weight_i1*, second input variable corresponding to the Weight element, and (iii) *unit_o0*, first output variable corresponding to the unit element. Types of these variables are inferred from imported schemas and example data files. The types are then passed instead of the generic type *T* in *Value<T>* (cf. Fig. 2). The Script dialog in Fig. 9 provides an example of the script specification. This transformation rule should be executed only when a value of the Weight column is greater than 7000 in the same CSV row.

In the Script dialog, developers can specify whether an operator is executed only once or multiple times for each input document. This can be done by checking or unchecking the *"Executed once?"* checkbox in the Script dialog. Operators can be executed multiple times for each input document in cases where a single document is divided into smaller data units. A data unit refers to an atomic piece of data that is

provided as a single input to a transformation system. For instance, in the CSV TS, each row of the payload can be sent independently and transformed to a desired target structure. However, in many cases, an XML document must be sent to be properly interpreted and transformed.

In the Script dialog, a developer can select an operator from a *Parent operator* combo box and set it as the parent to the current operator. This is necessary when an operator's execution depends on another previously specified operator (parent). By creating operator hierarchies, a developer can specify groups of operators executed in a particular order.

We should note that in Fig. 9, the manual specification step is denoted as 3a, covering the described process. On the other hand, 3b denotes mapping specification using the automation facility and is presented in Section 5.1.

Step 4: Invoke the microservice code generator to generate the microservice code. Upon completion, the mapping specification will serve as input to the Generator module, which generates an executable integration adapter.

Step 5 To initiate the integration adapter, it may effortlessly be launched within the execution environment. The adapter will then read the input data through the CSV binder and apply the relevant transformations based on the generated rules. Once the data is transformed, it is written to the target TS. To ensure that the data is written in correctly, the adapter utilizes the XML binder in compliance with the specified XML schema.

The left-hand side of Fig. 10 presents the XML document that contains the values generated after executing the generated code on top of the CSV file. To avoid repetition, we have excluded lines with the same structure as those shown in lines 4-8. The values are generated from the columns of the CSV document displayed in the top section of Fig. 8. The information system can use the generated XML document to create a line chart with the JSChart library. The right-hand side of Fig. 10 shows an example of this chart.

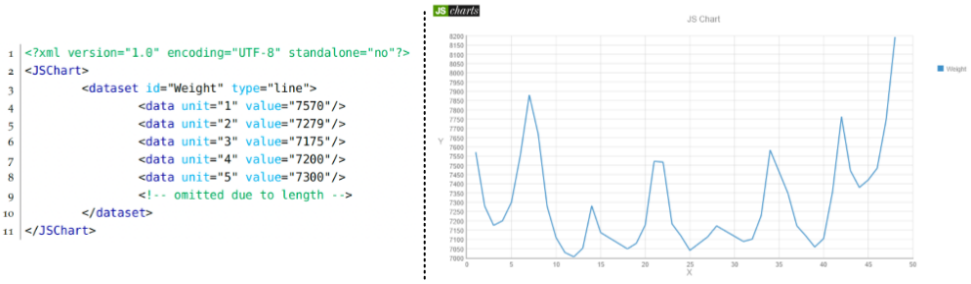

Fig. 10. The output XML file (left) and the line chart (right) of the single-layered measurement data

5.1. Automating the Device-to-information-system Integration

In Industry 4.0, variations in data schema often arise within the same integration scenario. This can result from using devices from different vendors or different

operation modes of the same device. Adapting existing adapters or creating new ones to handle these variations can be laborious and error prone. However, our approach provides a more efficient and user-friendly solution, as illustrated in this subsection.

In certain scenarios, the measurement device in wafer production employs a second measuring method as it is considered preferable and potentially ensures higher production quality, but at the cost of network utilization since it requires more data to be sent. Devices utilizing this measurement method require different configuration than those using the first measurement method. This configuration change necessitates the development or adaptation of an integration adapter, which can recognize the new data structure and transform it into the desired target technical space.

The new CSV document being sent is called a double-layered CSV, and an example is shown in Fig. 11. The wafer's weight is represented by columns titled Weight_A and Weight_B, among others. Some column names have been abbreviated in Fig. 11 due to space limitations. The full schema element names in the AnyMap tool screenshots are provided. In our use case, we require mapping double-layered CSV documents onto the same XML schema shown in Fig. 8.

Meas. Nr.	Ord. Nr.	Weight_A	Radius_A	Th._A	Pos.Pass_A	Fall.T.	Error_A	Fall._A	Weight_B	Radius_B	Th._B	Pos.Pass_B	Fall._B	Error_B
91	1	7041	6256	8	0.089	0	-9999	0	7113	6323	8	-0.182	0	-9999
91	2	6816	6059	8	-0.073	0	-9999	0	6839	6079	8	-0.091	0	-9999
91	3	6762	6014	8	-0.554	0	-9999	0	6756	6007	8	-0.062	0	-9999
91	4	6659	5926	8	-0.844	0	-9999	0	6638	5906	8	-0.058	0	-9999
91	5	3711	5314	3	10000	R	-9999	R	6621	5891	8	-47.542	U	-9999
91	6	6640	5912	8	0.051	0	-9999	0	6634	5909	8	-0.147	0	-9999
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Fig. 11. Double-layered CSV document example

There are a few ways to specify a mapping for integration. The first option is a manual specification, where the developer can specify the mapping from scratch, as described in Step 3. The second option is a semi-automatic specification, where the developer can rely on a mapping repository and automation algorithms. They can provide a set of element mapping candidates, and the developer can select the most suitable ones for the current integration task. This approach requires some manual intervention from the developer, but the degree of involvement is far lower than with the manual specification. The third option is an automatic specification. This requires the least amount of user involvement, as all tasks are performed automatically by the AnyMap tool based on predefined heuristics. However, for the automatic specification to yield satisfactory results, a mapping repository must be populated with enough mappings from that domain to steer the process in the right direction.

As we want to present the automation process in our double-layered example, we will present the mapping specification that follows the semi-automatic approach and uses the reuse automation algorithm.

To initiate the mapping specification process, the first step is to load the double-layered CSV and XML schemas into the AnyMap tools. Once both schemas are loaded, the developer selects the reuse automation mechanism and sets it up through a dialog box on the left side of Fig. 12.. This can be an alternative to Step 3, as described earlier.

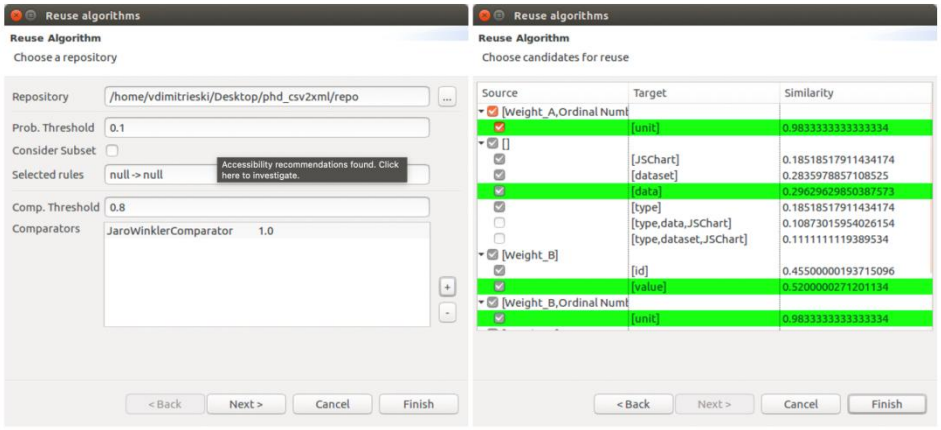

Fig. 12. The reuse configuration (left) and the element mapping candidates (right) dialogs

The first section of the dialog box defines the path to the reuse repository. In this example, a local repository is used. In the second section of the dialog box, the developer sets the probability threshold that defines a minimum probability at which an element mapping candidate is presented to the developer. If the *Consider Subset* checkbox is checked, the algorithm will provide element mapping candidates comprising a subset of schema elements the developer selects. Otherwise, only a superset or an equal set to the one provided as an input is considered a candidate. Selected schema elements provided as input to the algorithm are presented in the Selected rules textbox.

The third section of the dialog box allows the selection and configuration of schema matchers (comparators). Schema matchers are used to detect similarities between current schema elements and repository schema elements. These similarities are later used by the reuse algorithm and combined into a probability of a specific element mapping being appropriate for the current mapping context. Schema matchers can detect element similarity based on the structure or semantics of the elements.

In our particular use case, the data sent by the sensors follows strict naming rules. If a property is measured multiple times by a single sensor in a single measurement pass, a new letter is appended to the property name for each measurement. Therefore, the similarity between CSV columns (source schema elements) in this use case can be measured using a string comparison mechanism. In Fig. 12, the Jaro-Winkler [45] string comparison algorithm is selected as it is the most suitable for short names and strings.

AnyMap tool enables developers to select multiple schema matchers and use them simultaneously. Each selected matcher is assigned a weight value, which modifies the result of each comparator. In Fig. 12 there is one matcher, and its weight is set to 1.0. Additionally, the developer can set a Comp. threshold value, which determines the minimum similarity required for a matcher to report two elements as similar. In this example, elements are considered similar if they have a similarity score of 80% (0.8) or higher.

Once the configured reuse algorithm is executed, a dialog appears on the right side of Fig. 12, which displays the calculated element mapping candidates. These candidates are grouped by the source schema elements. Each group presents target schema

elements of the element mapping candidates and their probability of being a suitable fit for the current integration scenario. The element mapping candidates highlighted in green have the highest probability.

The developer chooses the most appropriate element mapping candidates for the current integration scenario. If an automatic algorithm were used, the candidates highlighted in green would be automatically selected. In either case, the selected candidates are applied automatically in the mapping canvas. The resulting mapping specification is presented in Fig. 13. The element trees on the left and right sides of the mapping canvas are created from the double-layered CSV document shown in Fig. 11 and the XSD document presented in the bottom part of Fig. 8, respectively. Compared to Fig. 9, we can observe that more mappings are created, representing the same relationships but, in this case, twice as the CSV has two layers of measurements.

The generated integration adapter takes the CSV document from Fig. 11 as input and produces the XML document from the left-hand side of Fig. 14 as output. We have omitted repetitive lines. Once the information system receives the XML document, it presents the content as a line graph, as shown on the right-hand side of Fig. 14.

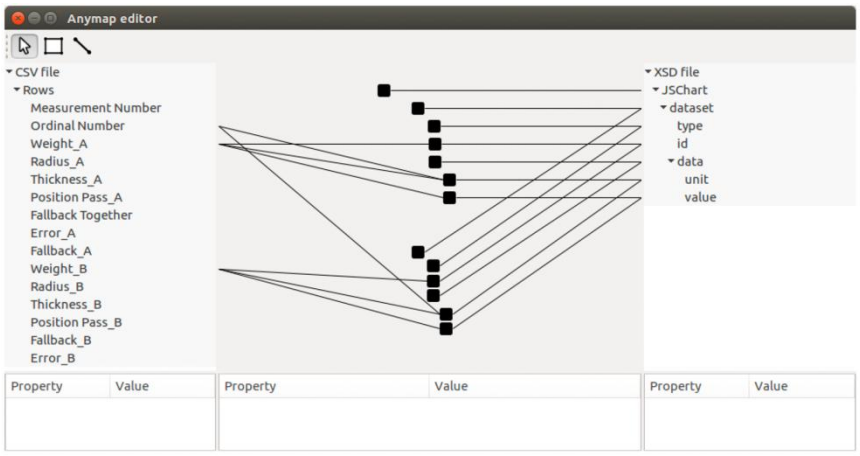

Fig. 13. The double-layered CSV2XML mapping specification

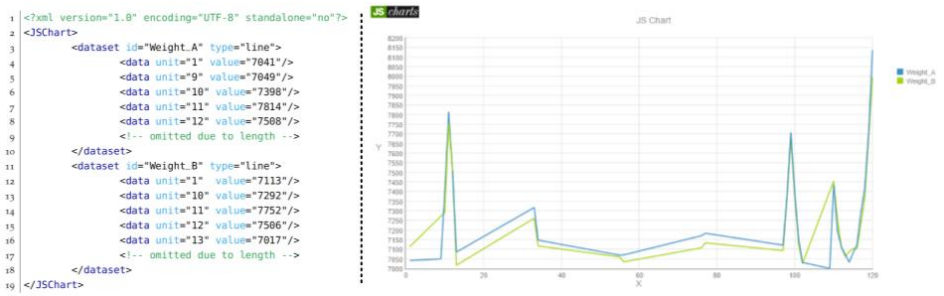

Fig. 14. The output XML file (left) and the line chart (right) of the double-layered measurement data

6. Conclusion

In this paper, we propose an approach to specifying and implementing integration adapters for any pair of TSs. Our approach employs a common data format to facilitate integration across diverse system components. Initially, we established a universal data structure onto which all other data formats can be mapped based on the core principles of MDSD. Each TS involved in integration is represented as a generic data model. Subsequently, integration experts and developers create transformations, i.e., mappings, using a declarative mapping DSML with a graphical notation. This allows the connection of structures from the source and target TS models, thus representing required mappings between them. Ultimately, the integration adapter is automatically generated from the mappings, leading to expedited development and reduced errors.

One notable aspect of this approach involves reusing existing mappings and creating new mappings based on those stored in a repository. Including knowledge and derivation functionality also allows for the automated specification of a new mapping. This reusability accelerates the integration process and simplifies the development process for integration engineers.

We have developed an integration tool called AnyMap to support our approach. This tool consists of the following components: TS importers (binders), a DSML for the specification of mappings, alignment and reuse algorithms, and code generators. AnyMap allows integration developers to specify mappings between systems at a higher level of abstraction and then use code generators to create executable adapters for a specific execution environment. In addition to the AnyMap tool, we have also developed a microservice-based scalable execution environment in which adapters are executed. However, creating other code generators that will generate regular, stand-alone Java or C adapters is possible. We have successfully applied our integration approach to integrate devices communicating via CSV with ISs communicating via XML.

The tool can be very useful to various software engineers and experts from different domains who need to integrate arbitrary TSs and do not have an adequate mechanism for mutual data exchange. With the tool, users can integrate arbitrary TSs uniformly, allowing users to learn the approach and become accustomed to the tooling support once and afterward, just spending time on performing the integration tasks without learning the implementation and serialization details of each one. AnyMap allows users to specify integration adapters at the level of data schemas using a custom DSML. As concepts of such a language on a highly abstraction level are tailored to non-programmers' needs and skills, by using these concepts, a user does not need to be experienced in any of the contemporary programming languages. Also, the reuse and alignment algorithms offered by AnyMap alleviate a user of the tedious, error-prone, and time-consuming process of creating repetitive mapping (transformation) rules.

Our approach has a limitation, as it can only be applied to three-level meta-models with a graph structure that can be transformed into a tree without losing significant semantic properties. This restricts its utility to schemas that can be flattened, i.e., schemas without infinite recursions of their elements. However, in our experience, we have seldom encountered infinite recursion in the cases where we have used our approach. Moreover, in instances where infinite recursion has been encountered at the schema level, flattening the structure to a desired depth and breaking the recursion with stub elements has proven to be an effective solution. We plan to address this limitation as a part of future work by investigating schema representation approaches and their implementation possibilities.

One potential avenue for future research is the development of more advanced and effective element-matching algorithms. At present, automation relies on simple matching algorithms that gauge element similarity and probability based on individual element information without considering the context or relationships of the element when determining the probability of it being the right candidate for a new mapping. An enhancement could involve semantically describing TSs using a shared ontology or a set of ontologies and automatically identifying semantic correspondences between source and target elements using ontology alignment methods.

We also plan to establish metrics and an evaluation framework for qualitative and quantitative assessment of our approach in future research endeavors. The research

community widely recognizes [46, 47] that evaluating an MDSD approach and measuring the quality of DSML are challenging tasks, and to date, only a limited amount of research has been conducted on these issues.

To advance the development of the AnyMap tool, it is essential to implement various enhancements to improve its usability, efficiency, and domain coverage. Introducing new TS binders would expand the tool's applicability across various use cases. This, in turn, would lead to a greater number of mappings in the reuse repository, fostering improved mapping automation. Our strategy involves developing binders for the most widely used protocols in the Industrial IoT (IIoT) domain, such as Open Platform Communications Unified Architecture (OPC UA), Message Queue Telemetry Transport (MQTT), and Modbus. With the addition of these protocols to the existing support for SEMI Equipment Communications Standard/Generic Equipment Model (SECS/GEM), CSV, and XML, we will achieve comprehensive coverage for many common use cases in the industrial environment.

Furthermore, we intend to enhance the tool by integrating a real-time execution engine. This functionality will allow users to promptly execute specified mappings and observe the results directly within the tool. This improvement will enhance debuggability and shorten the feedback loop. Additionally, expanding the tool with the new concrete syntaxes and allowing users to customize the graphical syntax for specifying the mappings could help increase the developers' efficiency.

We also need to expand our generator and execution environments. Although our current microservice execution environment offers the necessary flexibility and scalability, there are industrial settings where the speed of transformation execution is crucial. We intend to create a generator for optimized adapters using C or C++ programming languages to address this. These adapters will be executed directly on the industrial PCs connected to the devices. Running near the machines, these adapters will enable real-time or near-real-time data transformation. It's important to note that optimizing these adapters is crucial due to the limited resources on which they often run.

An additional enhancement would involve upgrading the tool to enable the transformation of streaming data, as opposed to data transferred in files. This enhancement would necessitate a comprehensive repository of predefined mappings within the domain of the receiving data stream. The tool would need to operate in a "headless" manner, without any GUI elements, and in a fully automated mode that does not require any user intervention. It would apply existing mappings to all compliant data packages within a stream, thereby promptly producing the output data. In the event that new, unknown variations of input data are encountered, for which suitable mappings are not already executed within the tool, the tool would autonomously create new mappings using alignment and reuse algorithms. These newly created mappings would then be applied to the input data, ensuring uninterrupted stream processing.

Acknowledgements. This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-65/2024-03/200156), the Faculty of Technical Sciences, University of Novi Sad, through project “Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad” (No. 01-3394/1) and by Faculty of Organizational Sciences, University of Belgrade.

References

1. Pourmirza, S., Peters, S., Dijkman, R., Grefen, P.: BPMS-RA: A Novel Reference Architecture for Business Process Management Systems. ACM Trans. Internet Technol. 19, 1–23 (2019). https://doi.org/10.1145/3232677.
2. Todorovic, N., Vjestica, M., Todorovic, N., Dimitrieski, V., Lukovic, I.: A Novel Approach and a Language for Facilitating Collaborative Production Processes in Virtual Organizations Based on DLT Networks. In: Proceedings of the 2nd International Conference on Innovative Intelligent Industrial Production and Logistics (IN4PL 2021). pp. 197–208. , Online (2021). https://doi.org/10.5220/0010720900003062.
3. Vještica, M., Dimitrieski, V., Pisarić, M., Kordić, S., Ristić, S., Luković, I.: Production processes modelling within digital product manufacturing in the context of Industry 4.0. Int. J. Prod. Res. 61, 6271–6290 (2023). https://doi.org/10.1080/00207543.2022.2125593.
4. Shamsuzzoha, A., Toscano, C., Carneiro, L.M., Kumar, V., Helo, P.: ICT-based solution approach for collaborative delivery of customised products. Prod. Plan. Control. 27, 280–298 (2016). https://doi.org/10.1080/09537287.2015.1123322.
5. Juric, M.B. ed: SOA approach to integration: XML, Web services, ESB, and BPEL in real-world SOA projects. Packt Publ, Birmingham (2007).
6. Hermann, M., Pentek, T., Otto, B.: Design principles for Industrie 4.0 scenarios: a literature review. Technische Universität Dortmund, Dortmund (2015).
7. Linthicum, D.S.: Enterprise application integration. Addison-Wesley Professional (2000).
8. Wortmann, A., Barais, O., Combemale, B., Wimmer, M.: Modeling languages in Industry 4.0: an extended systematic mapping study. Softw. Syst. Model. 19, 67–94 (2020). https://doi.org/10.1007/s10270-019-00757-6.
9. Pulier, E., Taylor, H.: Understanding enterprise SOA. Manning Greenwich, Conn (2006).
10. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, San Rafael (2012).
11. Mohamed, M.A., Challenger, M., Kardas, G.: Applications of model-driven engineering in cyber-physical systems: A systematic mapping study. J. Comput. Lang. 59, 100972 (2020). https://doi.org/10.1016/j.cola.2020.100972,.
12. Sebastián, G., Gallud, J.A., Tesoriero, R.: Code generation using model driven architecture: A systematic mapping study. J. Comput. Lang. 56, 100935 (2020). https://doi.org/10.1016/j.cola.2019.100935.
13. de Araújo Silva, E., Valentin, E., Carvalho, J.R.H., da Silva Barreto, R.: A survey of Model Driven Engineering in robotics. J. Comput. Lang. 62, 101021 (2021). https://doi.org/10.1016/j.cola.2020.101021.
14. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. CSUR. 37, 316–344 (2005).
15. Kelly, S., Tolvanen, J.-P.: Domain-specific modeling: enabling full code generation. John Wiley & Sons (2008).
16. Kern, H., Stefan, F., Dimitrieski, V., Čeliković, M.: Mapping-Based Exchange of Models Between Meta-Modeling Tools. In: Proceedings of the 14th Workshop on Domain-Specific Modeling. pp. 29–34. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2688447.2688453.
17. Kern, H., Stefan, F., Fähnrich, K.-P., Dimitrieski, V.: A Mapping-Based Framework for the Integration of Machine Data and Information Systems. In: Proceedings of 8th IADIS International Conference on Information Systems 2015. pp. 113–120. International Association for Development of the Information Society, Madeira, Portugal (2015).
18. Bellahsene, Z., Bonifati, A., Rahm, E., others: Schema matching and mapping. Springer (2011).
19. Ten Cate, B., Kolaitis, P.G., Tan, W.-C.: Schema mappings and data examples. In: Proceedings of the 16th International Conference on Extending Database Technology. pp. 777–780. ACM (2013).

20. Agreste, S., De Meo, P., Ferrara, E., Ursino, D.: XML matchers: approaches and challenges. Knowl.-Based Syst. 66, 190–209 (2014).
21. Bernstein, P.A., Melnik, S., Petropoulos, M., Quix, C.: Industrial-strength schema matching. ACM SIGMOD Rec. 33, 38–43 (2004).
22. Bernstein, P.A., Madhavan, J., Rahm, E.: Generic schema matching, ten years later. Proc. VLDB Endow. 4, 695–701 (2011).
23. Raghavan, A., Rangarajan, D., Shen, R., Gonçalves, M.A., Vemuri, N.S., Fan, W., Fox, E.A.: Schema mapper: a visualization tool for DL integration. In: Digital Libraries, 2005. JCDL'05. Proceedings of the 5th ACM/IEEE-CS Joint Conference on. pp. 414–414. IEEE (2005).
24. Alexe, B., Tan, W.-C.: A New Framework for Designing Schema Mappings. In: Tannen, V., Wong, L., Libkin, L., Fan, W., Tan, W.-C., and Fourman, M. (eds.) In Search of Elegance in the Theory and Practice of Computation. pp. 56–88. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-41660-6_4.
25. Golshan, B., Halevy, A., Mihaila, G., Tan, W.-C.: Data Integration: After the Teenage Years. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 101–106. ACM, Chicago Illinois USA (2017). https://doi.org/10.1145/3034786.3056124.
26. Duchateau, F., Bellahsene, Z.: YAM: A Step Forward for Generating a Dedicated Schema Matcher. In: Hameurlain, A., Küng, J., and Wagner, R. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems XXV. pp. 150–185. Springer Berlin Heidelberg (2016). https://doi.org/10.1007/978-3-662-49534-6_5.
27. Duchateau, F., Bellahsene, Z., Coletta, R.: A flexible approach for planning schema matching algorithms. In: On the Move to Meaningful Internet Systems: OTM 2008. pp. 249–264. Springer (2008).
28. Lee, Y., Sayyadian, M., Doan, A., Rosenthal, A.S.: eTuner: tuning schema matching software using synthetic scenarios. VLDB Journal— Int. J. Very Large Data Bases. 16, 97–122 (2007).
29. Büttner, F., Bartels, U., Hamann, L., Hofrichter, O., Kuhlmann, M., Gogolla, M., Rabe, L., Steimke, F., Rabenstein, Y., Stosiek, A.: Model-driven standardization of public authority data interchange. Sci. Comput. Program. 89, 162–175 (2014). https://doi.org/10.1016/j.scico.2013.03.009.
30. Kutsche, R., Milanovic, N., Bauhoff, G., Baum, T., Cartsburg, M., Kumpe, D., Widiker, J.: BIZYCLE: Model-based Interoperability Platform for Software and Data Integration. In: Proceedings of the MDTPI at ECMDA (2008).
31. Milanovic, N., Cartsburg, M., Kutsche, R., Widiker, J., Kschonsak, F.: Model-Based Interoperability of Heterogeneous Information Systems: An Industrial Case Study. In: Paige, R.F., Hartman, A., and Rensink, A. (eds.) Model Driven Architecture - Foundations and Applications. pp. 325–336. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_24.
32. Milanovic, N., Kutsche, R., Baum, T., Cartsburg, M., Elmasgünes, H., Pohl, M., Widiker, J.: Model&Metamodel, Metadata and Document Repository for Software and Data Integration. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Völter, M. (eds.) Model Driven Engineering Languages and Systems. pp. 416–430. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-87875-9_30.
33. Vuković, Ž., Milanović, N., Bauhoff, G.: Prototype of a Framework for Ontology-aided semantic conflict resolution in enterprise integration. In: Proceedints of 5th International Conference on Information Society and Technology. Society for Information Systems and Computer Networks, Kopaonik, Serbia (2015).
34. Vuković, Ž., Milanović, N., Vaderna, R., Dejanović, I., Milosavljević, G.: SAIL: A Domain-Specific Language for Semantic-Aided Automation of Interface Mapping in Enterprise Integration. In: On the Move to Meaningful Internet Systems: OTM 2015 Workshops. pp. 97–106. Springer (2015).

35. Luković, I., Čeliković, M., Kordić, S., Vještica, M.: An Approach to the Information System Conceptual Modeling Based on the Form Types. In: Karagiannis, D., Lee, M., Hinkelmann, K., and Utz, W. (eds.) Domain-Specific Conceptual Modeling. pp. 589–614. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-93547-4_26.
36. Čelikovic, M., Luković, I., Aleksić, S., Ivančević, V.: A MOF based meta-model and a concrete DSL syntax of IIS*Case PIM concepts. Comput. Sci. Inf. Syst. 9, 1075–1103 (2012). https://doi.org/10.2298/CSIS120203034C.
37. Walsh, E., O'Connor, A., Wade, V.: The FUSE domain-aware approach to user model interoperability: A comparative study. In: Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on. pp. 554–561. IEEE (2013).
38. Kern, H., Stefan, F., Dimitrieski, V.: Intelligent And Self-Adapting Integration Between Machines And Information Systems. Iadis Int. J. Comput. Sci. Inf. Syst. 10, 47–63 (2015).
39. Dimitrieski, V., Čeliković, M., Igić, N., Kern, H., Stefan, F.: Reuse of Rules in a Mapping-Based Integration Tool. In: Intelligent Software Methodologies, Tools and Techniques. pp. 269–280. Springer, Naples, Italy (2015). https://doi.org/10.1007/978-3-319-22689-7.
40. Črepinšek, M., Mernik, M., Bryant, B.R., Javed, F., Sprague, A.: Inferring context-free grammars for domain-specific languages. Electron. Notes Theor. Comput. Sci. 141, 99–116 (2005).
41. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: VLDB. pp. 49–58 (2001).
42. Djukić, V., Popović, A.: Handling complex representations in visual modeling tools for MDSD/DSM by means of code generator languages. J. Comput. Lang. 75, 101208 (2023). https://doi.org/doi.org/10.1016/j.cola.2023.101208.
43. Fowler, M.: Domain-specific languages. Pearson Education (2010).
44. Djukić, V., Popović, A., Luković, I., Ivančević, V.: Model variations and automated refinement of domain-specific modeling languages for robot-motion control. Comput. Inform. 38, 497–524 (2019).
45. Winkler, W.E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (1990).
46. Ozkaya, M., Akdur, D.: What do practitioners expect from the meta-modeling tools? A survey. J. Comput. Lang. 63, 101030 (2021).
47. Kahraman, G., Bilgen, S.: A framework for qualitative assessment of domain-specific languages. Softw. Syst. Model. 1–22 (2013).

Vladimir Dimitrieski works as an associate professor at the University of Novi Sad, Faculty of Technical Sciences, Serbia. There, he went through all the academic education levels, receiving a Ph.D. in computer science in 2018. Currently, he is a lecturer in several courses at the Faculty of Technical Sciences that cover domains of (meta-) modeling, domain-specific languages, and data engineering. With a strong background in the domain of Industry 4.0, (meta-)modeling and data engineering, he has been part of multiple national, international, and industrial projects in these domains.

Slavica Kordić received her M.Sc. degree from the Faculty of Technical Sciences, at University of Novi Sad. She completed her Mr (2 year) and Ph.D. degrees, both from Faculty Technical Sciences, University of Novi Sad. Currently, she works as an associate professor at the Faculty of Technical Sciences at the University of Novi Sad, where she lectures in several Computer Science and Informatics courses. Her research interests are related to Information Systems, Database Systems, Business Intelligence Systems, Data Reengineering and Model Driven Software Engineering. She is the

author or co-author of over 60 papers, and several industry projects and software solutions in the area.

Sonja Ristić works as a full professor at the University of Novi Sad, Faculty of Technical Sciences, Serbia. She received her bachelor's degree (Hons.) in Mathematics from the Faculty of Science, University of Novi Sad, Serbia, in 1983. The bachelor's degree (Hons.) in Economics, M.Sc degree in Informatics (former Mr, two years) and Ph.D degree in Informatics she received from the Faculty of Economics, University of Novi Sad, in 1989, 1994, and 2003, respectively. From 1984 until 1990 she worked with the Novi Sad Cable Company NOVKABEL – Factory of Electronic Computers. From 1990 till 2006 she was with Novi Sad School of Business, and since 2006 she has been with the University of Novi Sad, Faculty of Technical Sciences. Her research interests include Database Systems, Software Engineering, Model Driven Engineering, Domain Specific Languages, and production system modelling in the context of Industry 4.0/Industry 5.0. She is the author or co-author of over 100 papers, and 10 industry projects and software solutions in the area.

Heiko Kern receiving a Ph.D. in computer science in 2016. His research interests include Model-Driven Engineering, especially modeling and meta-modeling, model transformations as well as tool interoperability. He has experience in national and international funded research projects. Currently, he works as senior researcher at Institute of Applied Informatics at University of Leipzig, Germany and is head of the research group Software and System Integration.

Ivan Luković received his diploma degree (5 years) in Informatics from the Faculty of Military and Technical Sciences in Zagreb in 1990. He completed his M.Sc (former Mr, 2 years) degree at the University of Belgrade, Faculty of Electrical Engineering in 1993, and his Ph.D. at the University of Novi Sad, Faculty of Technical Sciences in 1996. Currently, he works as a Full Professor at the Faculty of Organizational Sciences of the University of Belgrade, where he lectures in several Computer Science and Informatics courses. His research interests are related to Database Systems, Business Intelligence Systems, and Software Engineering. He is the author or co-author of over 200 papers, 4 books, and 30 industry projects and software solutions in the area. He created a new set of B.Sc. and M.Sc. study programs in Information Engineering, i.e., Data Science, at the Faculty of Technical Sciences. The programs were accredited for the first time in 2015. Currently, he is a chair of the M.Sc. study program in Information Engineering at the Faculty of Organizational Sciences, as well as the chair of the Managing Board of the Computer Science and Information Systems (ComSIS) journal.

Received: July 01, 2024; Accepted: October 30, 2024.