Proceedings of Seminar

New Trends in Software Construction

2023

Editors: Horst Lichter
Ada Slupczynski
Selin Aydin
Alex Sabau

SWC Software Construction

RWTH AACHEN UNIVERSITY

Table of Contents

Ilija Kovacevic, Bastian Levartz:
How does software architecture correlate with legacy systems?

Joris Mohar, Konstantin Dao:
An Analysis of Graphical Notations for ML Solutions in Academic Research

Haydar Genc, Mohamed Amine Zormati:
Comparing Selected Security Modeling Languages Using SEQUAL Framework

Leila Mangonaux, Marco Heinisch:
Classification of modernization methods based on experience reports: A Systematic Literature Review

How does software architecture correlate with legacy systems?

Ilija Kovacevic
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ilija.kovacevic@rwth-aachen.de

Bastian Levartz
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
bastian.levartz@rwth-aachen.de

ABSTRACT

The problem of legacy software systems has been known for a long time and several approaches have been proposed to eliminate the issues concerning maintainability and modifiability. It can be achieved by transforming the current solution to a more maintainable one or decommissioning it after migrating to an entirely new system. However, any approach bears the risk of being time-consuming, costly and unsuccessful.

Therefore, it is of increasing interest to build software systems in a way that they are as maintainable and extensible as possible. This paper examines in an exploratory literature review what the connection is between software legacy systems and their software architecture. We look at the extent to which the maintainability, adaptability and performance of a legacy system is influenced by the chosen software architecture. In doing so, we also explicitly highlight architectures that either delay or advance a system in the process of turning into a legacy one.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

software architecture, legacy systems, architectural technical debt

1. INTRODUCTION

Reverse engineering, modernisation and migration of legacy software systems are a common practice. Many different approaches have been presented in the past years, approaching the issue from different perspectives [1, 7]. One thing that most of them have in common is a change of the architectural approach. For example in the last few years many monolithic systems had been migrated to a microservice architecture. Within the scope of this development it is a relevant question, whether the selection of an architecture correlates with the probability of creating a legacy system. Understanding the influence can not only enhance the selection of new architecture, but also offer opportunities to modify older, non-legacy systems in a way that slows down the process.

1.1 Definition of legacy systems

In the scientific community there is no uniform and fixed definition of legacy systems. However, there are some aspects in which most definitions agree. The main part characterising a system as legacy is its ability to deliver business value but simultaneously being accompanied with significant disadvantages. Those are the non-existing ability to change the legacy system in order to fulfill new business needs, the fragility of the system and lastly the increasing amount of necessary maintenance to keep it running [3, 4, 5, 18]. These properties also result in increasing costs for the persistence of the system.

To identify the possibility of systems becoming legacy we choose the following characteristics for the classification of a legacy system:

- High maintenance effort
- Significant business value
- Limited modifiability and expandability
- High integration efforts
- System is operative

Replacing such a system comes with technological, operational and organizational problems. These include missing documentation, unstructured code due to frequent modifications and missing knowledge from the original developers, because they retired or left the company [18]. In addition one has to find a suitable way to keep the legacy system running while engineering a new solution.

1.2 Overview Software Architectures

In any sufficiently complex system one has to use a global view outside of the systems inner workings in order to capture all necessary dependencies, core values and the human workflow. This encapsulates the definition of a software architecture as it represents the gross structure of a software system [30]. Furthermore, the architecture addresses the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2023 RWTH Aachen University, Germany.

business goals and quality goals by providing specific guidelines for reliability, performance, modifiability, usability and security [2].

We interpreted that a software architecture is the theoretical construct of how a business wants to develop complex pieces of software without specifying technology. We chose to examine shortly the history of how the software engineering landscape changed in the last few decades. This involved revisiting mainframes, the introduction of the internet, some scientifically examined architectures e.g. domain-driven or service-oriented architectures and the recent development towards microservices and cloud. These are the major architectures which are referenced as source or target ones in papers about migrations.

1.3 Problem statement

In an era of massive technological advances the pressure for continuous modernisation increases, making the adaption in the systems unavoidable. In case of software the handling of these changes is challenging, because a repetitive redevelopment is not possible due to costs and time. Therefore, the maintainability and portability is an important aspect of software engineering.

In this paper we discuss the potential correlation of the underlying software architecture of a system and its lifespan until it turns legacy. For this purpose we analyze certain software architectures and corresponding techniques for modernisation. In this context we also search for architecture models, which are designed to withstand the pressure of change and avoid or at least slow down the process of turning into a legacy system. More precisely, we defined the following research questions:

- **RQ1:** Does a correlation between software architectures and legacy systems exist?
- **RQ2:** Which software architectures increase the chances of a system becoming legacy?
- **RQ3:** Which software architecture are able to avoid or delay the creation of legacy systems?

1.4 Methodology

This paper presents an exploratory literature review on software architectures and their impact on the creation of legacy systems all over the world. For conducting this review we used the SCOPUS database. The review is split into two parts. The first one focuses on the term architectural technical debt, the second one on the correlation of legacy systems and their software architecture. The architectural technical debt part will be discussed in the third chapter and focuses on the definition of the term and corresponding problems caused. Furthermore, we have a look at presented approaches for the management of these problems. Within the scope of the literature review we searched in SCOPUS for resources with architectural technical debt in the title, the abstract or the keywords. We then excluded papers, which are not inside of the computer science area and papers with less than five citations.

The review for the fourth chapter is conducted similarly. We used the search string ”software AND legacy AND (modernisation OR migration)“. From the results we only picked papers written in English. Then we empirically analyzed the papers, by splitting them into several age groups and extracting the source and target architectures of the presented modernisations.

2. FROM MAINFRAME TO CONTAINER

The landscape of software and hardware architectures is ever changing. We examine how mainframe rose and fell out of popularity, the scientific advancement in software architectures and the most recent move towards cloud infrastructure and containerization. These are some major milestones which reflect critical changes in how businesses operate their software development.

2.1 Mainframe

When talking about legacy systems the most obvious example is a mainframe device as it is seen as the embodiment of a resource which is business critical but hard to migrate away from. It is by definition a large computer which used to fill a whole room but at this point the leading manufacturer IBM builds mainframes which are the size of a refrigerator [17].

In the time period from the first commercial mainframes in the 1950s until the 90s a mainframe was the only acceptable means of handle data processing for large business. They are built for reliability and a high throughput of transactions per seconds. This made them be at the core of any large business as they were utilized to process databases, costumer transactions or statistical analysis. All hardware components are redundant and have the ability of being exchanged while the system is running thus it enables a business to fully encapsulate all its business logic and computing needs on a single machine [17].

The software on a mainframe was mostly written in COBOL and as a procedural language which relied heavily on human readable keywords as such it was limited in what it could accomplish efficiently. The speed at which computer capabilities rose could not be reflected by COBOL. It was a language written in the style of punch cards in order to ease the transition to modern computers but this made it hard to write and follow the logic. There was no architecture designed to help with code re-usability, security or stability thus all COBOL code is a unique piece of business logic. With people retiring or leaving the knowledge of the code base shrunk and the corporations were bound to a physical legacy system which was unmaintainable [15].

2.2 Internet and distributed systems

The commercialization of the ARPANET and creation of the world wide web is regarded as the most influential technological milestones in human history. The availability of an internet connection has been declared as part of persons livelihood and an ISP has to carry that burden.

In the same time frame the computational power of PCs and small-scale Servers has reached a point which allowed a business to not invest in a mainframe. This lead to a movement towards the ”edge” of their infrastructure. A business was able to connect remote locations by using the third layer of the ISO/OSI model [23].

2.3 Software architectures

The starting point for modern software architecture can be contributed to the paper of Perry and Wolf [29] as it is referenced in most papers dealing with software architecture

[16, 30]. The paper lays the foundation of software architecture research. At its core a software architecture is the conjuction of "Elements, Form, and Rationale". Elements are divided into the categories processing, data and connection.

The architectural form can be understood as an indicator of how important a piece in the overall architecture is and in which relation it stands with all the other pieces. Finally the rationale is the underlying motivation of why certain choices were made [29].

2.3.1 Domain-driven architecture

Following the starting point of software architectures the first one which evolved into a large set of models, rules and guideline was the "domain-driven design" which is outlined by Eric Evans in 2003.

An architecture which follows the guidelines of domain-driven design will be founded on two principles:

- All software components reflect business and domain logic
- Any interaction between components is based on a model

These principles are not of technical nature but they advise how to think about the complex software project at hand. It is a way of translating the businesses knowledge into a suitable software product by combining the knowledge of developers and business analysts by following "model-driven designs". These entail the direct communication of developers and business analysts in order to translate the business needs directly into an abstracted software model.

The transformation to a working product is supported by technologies which are able to clearly represent this model. These are in most cases object oriented languages and simple architecture patterns e.g. layered models or monoliths.

The usage of a domain-driven design for a particular software product will result in an encapsulation of domain logic in a technical domain objects which needs to be (directly) accessed by a client to be modified. In front of such a request will be a gateway service which is responsible to deliver this request from the client to the domain object. This "gateway pattern" does not scale well in environments where the manipulation of domain objects needs to happen often, fast and regularly [10].

2.3.2 Service oriented architecture

The necessity to decouple business logic from specific objects resulted in the object agnostic way of architecting software. The service oriented architecture (SOA) is supposed to help developers and architects follow principles to accomplish that task.

One can find many definitions of SOA and its corresponding practices. We will draw on the summary done by Duggan [10] in order to gain an understanding how SOA is defined. It is outlined as a set of "design characteristics" and "design principles".

The characteristics include but are not limited to:

- Standardized service contract: Services adhere to a standard communications agreement, as defined collectively by one or more service description documents within a given set of services.
- Service reusability: Business logic is split into several services and is "context agnostic", to promote reuse of code.
- Service autonomy: Any Service can work independently thus each service can work autonomously even others are not working properly.
- Service statelessness: In a stateless service, each request is treated independently, without any reliance on previous requests or knowledge of the client's past interactions.

The design principles define *how* these design characteristics are applied:

- Service loose coupling: Services are designed to have minimal knowledge and reliance on each other, allowing them to evolve and operate independently without significant impact on other services.
- Service abstraction: It enables service consumers to interact with services without being aware of the underlying technologies, platforms, or complexities involved.
- Service composability: Services can be combined to create higher-level services or complex business processes.

A subset of further principles includes "service discoverability", "service statelessness" or "service autonomy" [10].

2.3.3 Microservices

The concept of microservices can be seen as a newer and revisioned version of SOA with key differences in the way these architectures are governed, the relative size of a service and the autonomy.

SOA emphasizes centralized governance and standardization of service contracts and interfaces. It often employs an "Enterprise Service Bus" or a central service repository for service management and orchestration. In contrast, microservices prioritize decentralized governance, allowing individual teams to have autonomy over their services without relying on a centralized infrastructure or standardization. This enables any team to individually deploy or scale their provided service [10].

Furthermore, a service needs to be "micro" to serve a business function in a microservice architecture which is not clearly defined in SOA. The service size may be small in a SOA software project but it is not prioritized [33].

The scientific community surrounding software engineering has had a major interest in microservice architecture as in recent almost all migrations have been moving towards such an architecture. It is regarded to be one of the most efficient ways of dealing with legacy systems as the process is well documented. Another advantage is that splitting up a layered monolith can happen in small batches and reduces risk of a vital business workflow failing [33].

The migration from legacy codebases towards microservices is often accompanied by a mindset change towards a more agile approach in software development. The modernisation towards such an architecture, aims to achieve a faster time to market of new software releases, high available systems which are more resilient against any types of failure, better maintainability of the codebase and a "cloud-ready" infrastructure [32].

2.4 Cloud / Container

We will briefly introduce cloud and container technologies in order to understand how these technologies are used as an integral component of software architectures. They are most often deployed in combination because the small footprint of a container can be used well in a cloud environment.

"The cloud is just someone else's computer" is a popular joke among people criticizing the movement to the cloud. One can differentiate between three services a cloud provider offers:

- IaaS: Infrastructure as a Service e.g. virtual machines, routers, load balancers
- PaaS: Platform as a Service, e.g. Databases, App Engines, Storage
- SaaS: Software as a Service, e.g. Mail, Netflix, Zoom

The overall benefit of any cloud offering is flexibility to scale up or down almost infinitely and the abstraction layer which handles underlying systems depending on what service one uses.

Containers are lightweight and isolated environments that package software applications and their dependencies, allowing them to run consistently across different computing environments. They provide a standardized way to package and distribute applications, making them portable and easy to deploy [27].

They are most suitable to be utilized in a microservice architectures because they can be run autonomously, can be scaled up or down independently and they represent a single business function [27].

3. TECHNICAL DEBT

The metaphor technical debt has been introduced by Ward Cunningham in 1992 in the beginning of agile development and has been spread further since, together with the growth of agile development [8]. It is used to describe a trade off between short-time and long-term value, which is for example induced by the pressure to meet a deadline[6, 26]. This comes at the cost of "debt" which has to be repaid and increases the total amount of engineering work [25]. The creation of technical debt in the process of writing new software is not necessarily bad, but causes problems, when it's not repaid promptly. Furthermore, technical debt increases the costs for maintenance during the life span of the application [8]. Using our defined characteristics for legacy systems the increasing maintenance effort can therefore increase the risk of a system to become legacy.

Another problem is, that technical debt can also be created within the lifetime of a software, introduced by bug fixes and continuous development of new features. In this context technical debt often refers to debt on the code level. For example structural problems in the code or the disregard of best practices[21].

Since the creation of the metaphor technical debt, the term has been used in many different endeavours, to describe different kind of debts. This includes for example code debt, documentation debt, architectural debt, requirement debt or infrastructure debt [19, 28].

Following from the architectural focus of the paper, we especially focus on architectural technical debt. As Ernst et. al showed in their empirical study, about the understanding and view on technical debt, that most of the debt results from poor architectural choices [13].

3.1 Architectural technical debt

Architectural technical debt (ATD) refers to a special form of technical debt, caused by poor architectural design decisions over the course of time. This includes violations of best practises and the integrity of the software architecture [22]. ATD can also be caused by the decisions to use already existing legacy components, instead of rewritting them from the beginning. The decision to create debt in such a way, is often driven by economic factors like time-to-market or financial costs [21].

The main problem of creating ATD in a system is the influence on the quality attributes of the system. Especially the ability to add new or change components (evolvabilty) and the maintenance gets harder with increasing debt, because the system deviates from the standard with every decision for a shortcut[21]. Therefore, also the costs rise, which can be a reason for companies to classify the system as legacy. The decision to take the debt may be better for the short term view, but the higher short term business value from the decision is destroyed by the effects in the long term costs and loss of quality[21].

A further problem of ATD results from the point in time, where architectural decisions are made. Most of the classic approaches to the development of software systems put the architectural decisions at the beginning of the process, which can make it hard to foresee all consequences of certain decisions[13]. This effect can increase when the business interests (limited time or budget) play a key role in the decision process. Ernst et. al also found out, that most of the developers they interviewed see the repaying of the debt as the bigger challenge. The long life span of a software system causes it to continuously drift further away from the original decision, design and also documentation[20]. This is underlined by their results, which include a proof for at least a small, measurable correlation between the age of a system and the notion of architectural issues by the developers [13].

A big problem in the scope of this development is the inability to calculate the value of paying back debt. The potential increases on overall system health, maintainability or the ability to simplify the construction of new features in the future is harder to estimate than the costs and gains of a completely new feature, visible to the customer [26]. This is especially the case in an agile development environment, because the agility favors shorthand decisions rather than a complete planing of the system before starting to program, as done in classical, linear approaches.

From the publications listed in SCOPUS the first year where the term ATD is considered, is 2012 and it became a relevant issue in the science community from the year 2018 on. This suggests, that the issue of debt in the architectural part of a system is a quite recent discovery, because the pure technical debt, which mostly focuses on problems in the source code, has already been an issue from 1992 on, where Cunningham defined the metaphor.

Since the appearance of architectural technical debt, the management and identification of it is a central issue in the research.

3.1.1 Managing architectural technical debt

Within the scope of our review we found multiple ap-

proaches to the management of ATD. These are spreading between simple approaches for the inclusion of found issues in the development process to more complicated concept models and metrics to estimate the current debt. This development has a high importance, because back in 2015 the study presented by Ernst et. al showed, that about 65% of the participants did not have a defined and systematic practice for the management of technical debt[13].

Overall the management of ATD can be separated into two parts. The first part describes the preventive management, which is used during the initial software development, with the target to decrease the amount of debt created and to ensure that the creation of debts is conscious. The second approach is the implicit handling of ATD, which focuses on the debt, that has already been created[21].

A simple approach for the implicit handling of technical debt, has been presented by Brown et. al. They proposed to handle technical debt in an agile environment, by simply adding the found issues to the product backlog, next to all the new features, which are included anyway [6]. This allows the reuse of already existing valuation methods from the classical project planing.

In addition Nord et. al presented a more systematic approach and created a formula to calculate the total development costs, based on splitting the costs into the direct development costs and the costs created in the rework, when deciding for taking on debt. The target is to make the decisions related to the value and offer an approach to handle debt within the development process [26]. In this case they described the management of technical debt as a "choice between focus on value and focus on costs", which results in agile methods tending to accumulate debt and more phase based approaches, focusing on cost, tending to delay the creation of value [26].

All the presented methods focus mainly on the implicit handling of ATD, which is quite important, because as mentioned in the introduction to technical debt, debt only becomes a real problem when it becomes too large. Therefore, it is a key part of the management to visualize the current debt level. Such a visualization has already been introduced for classical technical debt by Eisenberg et al. They defined different metrics that describe the amount of debt and then introduced thresholds so that timely action can be taken if the debt becomes too big [11]. This approach can probably also be fit to the architectural debt management, by defining different metrics and fitting thresholds.

Besides these implicit approaches it would be very useful, if approaches for the management exists, which aim to prevent the creation of debt. A possible approach for this has been presented with the development of Continuous Architecture. This approach mainly focuses on fitting the software development process to the acceleration of the development cycles, following from the trend of continuous delivery. One of the six key principles is to "delay design decisions until they are absolutely necessary"[12]. This could lead to the reduction of ATD, because the decisions are made with the most knowledge possible and are fitted as good as possible to the current system.

4. CORRELATION

Besides the architectural technical debt, which is created by many but small decisions about the architecture in the development process, the reference architecture could also have an influence on the probability of a system to turn legacy. This can be due to effects, which are created by decision made in the definition of the reference architecture or technological developments, which create needed abilities, not considered in the creation of the architecture.

Following from the focus of this paper, we do not analyze the effects induced by the development of new programming languages. Today no new system will be written in COBOL or Fortran, because many new programming languages are easier to understand and therefore more popular. In addition many of these old programming languages were developed in a time, where storage and computing power had been rare, resulting in different requirements for a language. Systems written in a dying language have to become legacy, because together with the language much of the knowledge about the language dies. Therefore, the costs for the people to do the maintenance and adjustments increases continuously, which is a characteristic of a system turning legacy.

Nevertheless the programming language cannot be excluded completely for the investigation of the relationship of the architecture and the life span of the software systems, since new programming languages also often bring new concepts, which form the basis for new architectures.

4.1 Empirical analysis

As described in the beginning, this part analyzes our research results from the SCOPUS database, by looking at the source and target architectures used in different migration or modernisation approaches presented in the different time spaces. We decided to divide the found documents into four equally sized time spaces. This results in the following time spaces: 1996-2002, 2003-2009, 2010-2016, 2017-2023.

Due to the short time space of the seminar we need to restrict the number of analyzed papers. Therefore, we only take the first 15 papers with the most citations from each time space and then filter out the paper that do not describe a migration or modernisation process of a software system.

4.1.1 1996-2002

In the time span from 1996 to 2002 nine out of fifteen papers contained a migration or modernisation of a legacy system in the software engineering space. The nine results focused primarily on transitioning from one language to another one. None of the papers mentioned any specific target architecture or any specific pattern which we could assign to one of our outlined architectures from chapter two.

source	target	count
COBOL / PL1	OO	5
C	OO	2
Assembler	C	1
Mainframe	Server-Client	1

Table 1: source and target of migrations from 1996 to 2002.

As can be seen in Table 1 most migrations targeted mainframe systems towards object-oriented languages. Some papers mentioned specific languages e.g. Java or C++ but most were focused on technical algorithms to extract information out of procedural languages into specific objects.

4.1.2 2003-2009

In the time span from 2003 to 2009 10 out of the 15 selected paper are helpful for our purpose. The other five had to be excluded, because they did not contain information about a system migration, but a specific health system or a visualization for high performance computing. 100 % of the useful papers described a migration to a SOA, that in some cases is specified as Webservice architecture. Based on what can be extracted from the papers they all start with a monolithic or formbased architecture. Both of these can be described as a domain-driven architecture. Two of the 10 approaches proposed a simple black box wrapping approach that takes the legacy system and wraps it into an architecture enabling the switch to a SOA.

4.1.3 2010-2016

From the 15 selected papers within this time span, only seven are helpful for our research, because the rest focuses on the evolution of GPUs or on the transition from classical network structures to the implementation of Software Defined Networking infrastructure. The useful papers can be divided into two parts. The first groups consists of three papers describing a migration to a SOA. The second group focuses on the migration of existing systems in to an cloud environment. The second group therefore mainly describes a change of the underlying hardware, which does not give much information about the influence of the software architecture, but the influence of technological change. However, it is possible to derive information from the migration, because the process can be divided into two different approaches. The first approach simply moves its legacy system to the cloud. In other words, they are only replacing the hardware. Thus, this approach does not really lead to an improvement in the issues characterising the system as legacy. The second approach includes not only the move to the cloud, but also a reengineering of the system. In this case SOA is still the chosen architecture, as for example in [9, 24].

4.1.4 2017-2023

Seven out of 15 papers specified the source and target architectures in order for us to analyze them further. Out of the eight remaining papers three were dealing with software-defined networks and the rest had miscellaneous content e.g. biology, quantum computing which is of no interest for our analysis.

The seven paper all dealt with the target architecture of microservices while the source architecture was always a monolithic design. Specifically the paper "Microservices Migration in Industry: Intentions, Strategies, and Challenges" [14] consisted of 14 different systems migrating from a monolith to multiple microservices.

The research interest in microservices started in 2015 [32] but is apparently dominating the field of migrations and modernisations.

4.1.5 Results

The most apparent result one can draw from our analysis is that until now the migration towards new software architectures has been acyclic. This means that the field of research and the software architecture in private companies has been moving in one direction only.

Furthermore, our results indicate that the modernization of software architectures schemes only became important from 2003 onwards. Before that the modernisation and migration papers only focused on a switch of the programming language.

One can see in Figure 1 that all of the identified architectures can be attributed to the overarching domain-driven design. While in Figure 2 one can see that a heavy focus lies on SOA and microservice architecture.

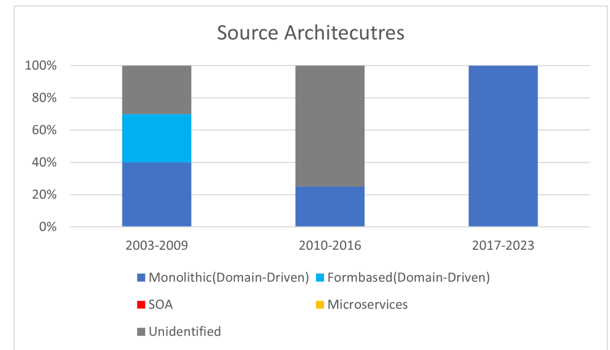


Figure 1: Source architectures for modernisation approaches from 2003-2023

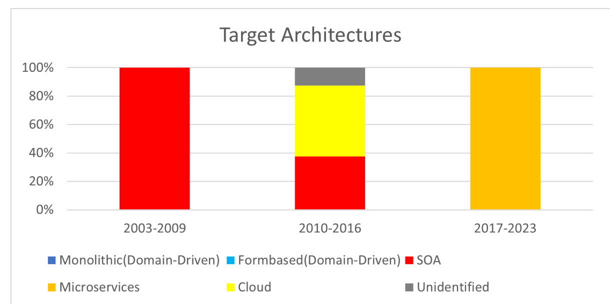


Figure 2: Target architectures for modernisation approaches from 2003-2023. The cloud characterisation has been added, because it is an important path within that timespace, although it does not directly corresponds to a software architecture. It can be added partwise to monolithic or to SOA as described in chapter 4.1.3

4.1.6 Methodology problems

The results of the literature analysis must be viewed with some caution, as our chosen procedure has problems in both internal and external validity. The internal validity is mainly affected by the lack of specificity in the publications. Most of the papers used say very little about the exact initial architecture as well as about the reasons that made modernization necessary. This is probably due to the fact that the approaches presented should be kept as general as possible and therefore not limited by more specific information. Overall, these problems mean that the learnings from the review are limited.

We also found some aspects in the external validity that limit the transferability of the results. The main issue here is that of the 60 papers selected, only a total of 33 were suitable for analysis, which results in a hit rate of 55%. This problem could have been reduced by a more thorough pre-selection of the papers. Furthermore, the search strategy could have

been improved by taking a certain number of papers for each year instead of looking at very long periods of time.

5. DISCUSSION

The software architecture is a major part of the development and research in software engineering. The goal is to create systems fulfilling quality attributes like maintainability, scalability, modularity and security, while also keep the overall costs for production and maintenance low.

An important part of keeping the costs down is trying to prevent the system from turning into a legacy system, because the reengineering and the new development is expensive. Therefore, we compiled the architectures and ecosystem which should be avoided or used.

5.1 Architectures to avoid

Regarding RQ2 we could extract two main architectures out of the learnings from the review, which should be avoided to improve the longevity of a system. The first architecture is based primarily on empirical analysis. We found that almost all modernisations move away from monolithic architectures to service-oriented architectures. The main reason for this is the lack of scalability of monolithic systems, as they offer several services, not all of which are used equally often. It follows that the services can not be independently scaled, which leads to a waste of resources for the services rarely used [31]. This effect can be intensified by the advancement of the Internet, which requires high scalability and flexibility. Therefore, the avoidance of monolithic systems can be a sensible step to slow down the process of turning legacy, especially for web applications offering multiple services.

Another practice to avoid is the multiple wrapping of legacy systems. In this case, the legacy system remains as a blackbox and is simply connected to the new system by a wrapping layer. This is a very low effort solutions, but comes with the problem that the issues of the actual legacy system, such as maintenance or central dependencies with limited support, are not fixed, but only obscured. Running this practice multiple times can make the system incomprehensible, increasing the cost of reengineering at some point in the future.

5.2 Architectures to choose

Answering RQ3 is complicated, because our research can not foresee the future. Therefore, we have to be careful about calling out architectures which will not turn obsolete in the distant future. We can however focus on the reasons why the migration or modernisation happened towards a specific architecture.

The migration towards SOA is supported by the development of modular and reusable services. By breaking down applications into smaller, independent services, organizations can reuse these services across multiple systems and applications. This modular approach enhances flexibility, scalability, and maintainability, as changes in one service do not necessarily impact the entire system [10].

Migrating to a microservice architecture has the same advantages while also having decentralized governance and scaling down the size of a service to a single function. Furthermore, the current technologies are driven by supporting such an architecture e.g. container, kubernetes or cloud [32, 33].

Moving infrastructure or software to the cloud is not strictly a migration of a software architecture but it still influences how the software products will be written. Cloud computing enables organizations to scale their resources based on demand, providing scalability and flexibility. This elasticity allows businesses to handle spikes in traffic, accommodate growing workloads, and optimize resource allocation [27].

6. CONCLUSION

Our exploratory literature review has partly succeeded in answering our research questions. We were able to find some correlations between architectures and their tendency to devolve into a legacy systems and architectures which have shown to be more resilient. In general one can say that monoliths, mainframes, procedural languages and variances of domain-driven architectures show a tendency to become unmaintainable.

The most common target architectures for modernisations were SOA, cloud infrastructures and microservices which is due to their flexibility, modular nature and scalability. In addition to the influence of reference architectures, the review also showed the impact that even small decisions, taken during or before the lifetime of a system, can have on the lifetime of the system, because they can appear as architectural technical debt.

The chosen methodology shows problems regarding the selection of publications thus future work should draw from multiple databases, refine the search terms and filter more heavily towards software architectures. Lastly one could focus on the impact new technologies have on longetivity of software systems. This may include DevOps practices, CI/CD pipelines, machine learning or advanced cloud configurations.

7. REFERENCES

[1] R. Arshad Khan. Trends in software reverse engineering. 04 2020.

[2] R. Bahsoon and W. Emmerich. Evaluating software architectures: Development stability and evolution. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, Tunis, Tunisia*, pages 47–56. IEEE Computer Society Press, 2003.

[3] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, 1995.

[4] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio. Iterative reengineering of legacy systems. *IEEE Transactions on Software Engineering*, 29(3):225–241, 2003.

[5] J. Bisbal, D. Lawless, B. Wu, J. Grimson, V. Wade, R. Richardson, and D. O'Sullivan. An overview of legacy information system migration. In *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, pages 529–530, 1997.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In G.-C. Roman and K. Sullivan, editors, *Proceedings of the FSE/SDP*

workshop on Future of software engineering research, pages 47–52, New York, NY, USA, 2010. ACM.

[7] S. Cetin, N. I. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu. A mashup-based strategy for migration to service-oriented computing. In *IEEE International Conference on Pervasive Services*, pages 169–172, 2007.

[8] W. Cunningham. The wycash portfolio management system. 1992.

[9] J. Delsing, J. Eliasson, R. Kyusakov, A. W. Colombo, F. Jammes, J. Nessaether, S. Karnouskos, and C. Diedrich. A migration approach towards a soa-based next generation process control and monitoring. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, pages 4472–4477. IEEE, 2011.

[10] D. Duggan. *Enterprise software architecture and design: Entities, services and resources*, volume 10 of *Quantitative software engineering series*. Wiley, Hoboken NJ, 2012.

[11] R. J. Eisenberg. A threshold based approach to technical debt. *ACM SIGSOFT Software Engineering Notes*, 37(2):1–6, 2012.

[12] M. Erder. *Continuous architecture : sustainable architecture in an agile and cloud-centric world*. Morgan Kaufmann, Amsterdam, [Netherlands, 1st edition edition, 2016 - 2016.

[13] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In E. Di Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 50–60, New York, NY, USA, 2015. ACM.

[14] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2019.

[15] Gary Barnett. The future of the mainframe. 2005.

[16] W. Hasselbring. Software architecture: Past, present, future. In V. Gruhn and R. Striemer, editors, *The Essence of Software Engineering*, pages 169–184. Springer International Publishing, Cham, 2018.

[17] C. Jacobi and C. Webb. History of ibm z mainframe processors. *IEEE Micro*, 40(6):50–58, 2020.

[18] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do professionals perceive legacy systems and software modernization? In P. Jalote, L. Briand, and A. van der Hoek, editors, *Proceedings of the 36th International Conference on Software Engineering*, pages 36–47, New York, NY, USA, 2014. ACM.

[19] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[20] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., USA, 1985.

[21] Z. Li, P. Liang, and P. Avgeriou. Architectural debt management in value-oriented architecting. In *Economics-Driven Software Architecture*, pages 183–204. Elsevier, 2014.

[22] Z. Li, P. Liang, and P. Avgeriou. Architectural technical debt identification based on architecture decisions and change scenarios. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 65–74. IEEE, 2015.

[23] T. C. Melewar and N. Smith. The internet revolution: some global marketing implications. *Marketing Intelligence & Planning*, 21(6):363–369, 2003.

[24] P. Mohagheghi and T. Sæther. Software engineering challenges for migration to the service cloud paradigm: Ongoing work in the remics project. In *2011 IEEE World Congress on Services*, pages 507–514. IEEE, 2011.

[25] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Foser 2010: The fse/sdp workshop on the future of software engineering research. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2010, New York, NY, 2010.

[26] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 91–100. IEEE, 2012.

[27] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019.

[28] E. Penttinen, N. Mäki, and T. Rinta-Kahila. *A Domino Effect: Interdependencies among Different Types of Technical Debt*. Honolulu, HI?, 2023.

[29] W. Perry. Foundations for the study of software architecture.

[30] M. H. Valipour, B. Amirzafari, K. N. Maleki, and N. Daneshpour. A brief survey of software architecture concepts and service oriented architecture. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 34–38. IEEE, 2009.

[31] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, 2015.

[32] H. Vural, M. Koyuncu, and S. Guney. A systematic literature review on microservices. In O. Gervasi, B. Murgante, S. Misra, G. Borruso, C. M. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, E. Stankova, and A. Cuzzocrea, editors, *Computational Science and Its Applications – ICCSA 2017*, volume 10409 of *Lecture Notes in Computer Science*, pages 203–217. Springer International Publishing, Cham, 2017.

[33] Z. Xiao, I. Wijegunaratne, and X. Qiang. Reflections on soa and microservices. In *2016 4th International Conference on Enterprise Systems (ES)*, pages 60–67. IEEE, 2016.

An Analysis of Graphical Notations for ML Solutions in Academic Research

Joris Mohar
RWTH Aachen University
Ahornstraße 55
52074 Aachen, Germany
joris.mohar@rwth-aachen.de

Konstantin Dao
RWTH Aachen University
Ahornstraße 55
52074 Aachen, Germany
konstantin.dao@rwth-aachen.de

ABSTRACT

This paper gives a guideline for graphical notations for future papers. We also form a clearer analogy for meta-models in software architecture to enhance understanding and communication between researchers.

There needs to be more prior research into graphical notation in machine learning (ML) solutions since little to no research on meta-models in ML in general exists. Many academic research papers use graphical notations that are difficult to understand and read. In this analysis, we explore why these notations are inadequate and which are of better use.

By applying Grounded Theory, we analyzed the different groups of graphical notations in academic research. We analyzed about 50 papers from the last 34 years to understand how these graphical notations work and evolve.

Throughout all the analyzed research papers, we discovered similarities and differences in flow charts and neural network (NN) diagrams. High-level overview components are often visualized as rectangles or circles and explained in other depictions. Furthermore, input data is included to improve the comprehensibility of how data is altered and used.

In conclusion, the trend goes from complex, difficult-to-read graphs to readable, straightforward graphs interacting with input data.

Keywords

Graphical Notations, Machine Learning Architecture, Graphs, Charts, Flow Charts, Neural Networks

1. INTRODUCTION

The Unified Modeling Language (UML) provides guidelines for conventional software architectures on how certain parts of the architecture should be represented. Despite efforts to visualize ML architectures for beginner ML practitioners [1] or to research in specific areas such as computer vision on the visualizations used [39], no similar guidelines exist for ML architectures in general.

Since ML practitioners find it difficult to present complex ML architectures in a way that is understandable to all stakeholders, there is a need for visualization or documentation guidelines [33]. Therefore we set up three objectives for this paper:

G1: First, we aim to provide a comprehensive collection of graphical notations used over the past 34 years.

G2: Our goal is also to offer a practical guide for best practices. We seek to alleviate the challenge of crafting effective scientific papers and enhance readers' understanding of architectural solutions.

G3: At last, we try to anticipate future trends.

The paper follows a clear structure. Section 2 explains our procedure, section 3 presents our findings and analysis, and Section 4 provides recommendations for using graphical notations and possible trends. Section 5 summarizes the paper's results, while section 6 includes acknowledgements.

2. METHODOLOGY

We analyzed these graphical notations in part through Grounded Theory [9]. We used this method by examining around 50 ML papers spanning 34 years and investigating which graphical notations they use to demonstrate the architecture of their solution.

The databases we searched through are Google Scholar, RWTH Publications [7] and arXiv.

The selection criteria included multiple keywords associated with subsections of ML, which were listed on the RWTH page from the AI Center [7]. The subsections were Deep Learning, Learning Theory, Learning on Graphs, Graph Neural Networks, Probabilistic and Statistical Learning, Reinforcement Learning, Data Mining, Process Mining and AutoML. We also added Supervised Learning and Unsupervised Learning to the selection criteria since they are also bigger subsections of ML [1]. Those criteria were often combined with keywords like graphical notations, visualization, model and architecture.

Initially, we added papers that lacked visual representations of the architecture to our collection. However, we later removed them from the collection due to their large number.

We skimmed through the visualizations for every search with applied selection criteria and added new research pa-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2022/23 RWTH Aachen University, Germany.

pers to the collection as long as the speed of new notations appearing has not decreased to an insignificant number.

After collecting the data, the relationships between the explored graphical notations were investigated, and similar notations were grouped under an overall theme.

3. RESULTS

Overall, five types of graphical notations were identified: automata, decision trees, nearest neighbor graphs, flow charts and NN diagrams.

We analyze in what cases these graphical notations are applied, as well as their advantages and disadvantages.

3.1 Automata

The first graphical notation we analyzed is automata, which illustrates the interactions and transitions between states in ML solutions.

Additionally, automata can effectively represent matrix states and associated data to give the reader a better idea of how the ML solution solves the problem. This is achieved through the numbering of all transitions, as it is obligatory to do for automata. It is mostly done by naming the transition by the numerals of the states it connects. A prime example demonstrating the interaction between automata and matrices is the Hidden Markov Model, which can be seen in figure 1.

However, automata have limitations in their applicability, as they are primarily suited for Hidden Markov Models but can be added as support for other graphical notations. Flow charts make more sense in most cases, as they allow for more detail and give a greater understanding of the ML solution than automata [16]. This advantage arises due to automata's limited capacity of representing only a limited portion of data along with its interactions and correlations [31].

Automata can become harder to interpret and comprehend as the ML solution grows in complexity and size. This absence of a mechanism to condense and manage data within automata and the limitation of representing only one data level pose a challenge to handling more extensive ML solutions.

In conclusion, while automata find their relevance in Hidden Markov Models, they have a very limited range of applications for other ML solutions, especially because of their constraints in representing data and their scalability issues [31].

3.2 Decision Trees

One commonly used way to visually represent machine learning solutions is through decision trees. They are most often used to show ML solutions' various decision paths. Additionally, decision trees give insight into probabilities and factors influencing the selection of the path they could go down or for what reasons they might take that path.

Decision trees excel at showing data effectively. Each node can give data about itself, including data about the nodes that come after or before them and their interactions.

However, decision trees pose certain challenges, especially in a research paper context, as we can only show a very limited number of decisions. The reason for that is the scaling with k^n, where k is the number of decisions possible for each node and n is the number of iterations of these decisions. For binary decision trees, which are the minimum

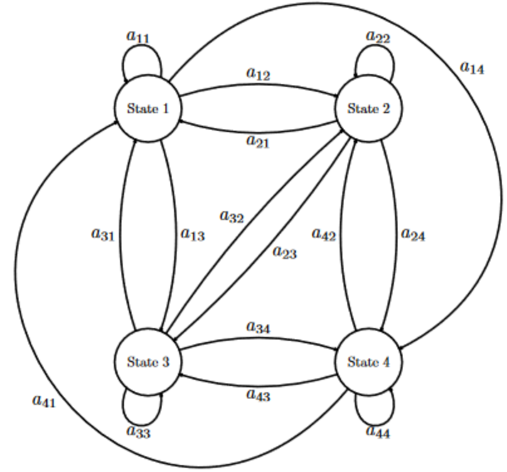


Figure 1: Automata. Taken From [31]

decision tree possible from a scaling perspective, the scaling effect becomes 2^n. Consequently, decision trees with more potential decisions show a more pronounced scaling effect.

Another problem is that decision trees primarily provide information about the decisions themselves. Only limited information about individual nodes and their immediate surroundings is conveyed. Consequently, as a standalone graphical notation, a decision tree often does not display a comprehensive overview of the ML solution, as seen in figure 2. However, it can effectively be used with other graphical notations like flow charts or heat maps to improve the overall comprehension of the ML solution [22].

In summary, decision trees are valuable for showing decision paths in an ML solution. However, their scalability issues and narrow focus often make combining them with other graphical notations necessary to offer a more comprehensive understanding of the ML solution [2] [38].

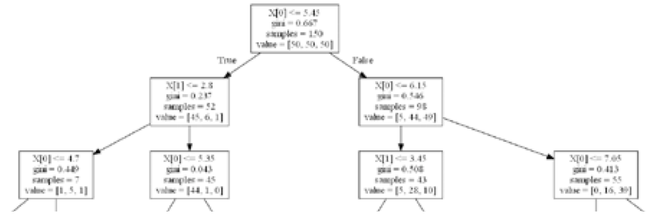


Figure 2: Binary Decision Tree. Taken From [2]

3.3 Nearest Neighbor Graphs

The third notation we analyzed was nearest neighbor graphs, which depict how an ML algorithm identifies nearby nodes and constructs clusters based on the nearest neighbor configuration. It uses multiple starting points to illustrate which nodes are closest to which respective starting point.

Nearest neighbor graphs, while powerful for nearest neighbor and clustering algorithms, have certain limitations. They provide little information about the ML solution and are confined in the application to nearest neighbor and clustering algorithms.

However, this specification helps to enhance their usefulness in showing nearest neighbor and clustering solutions as

well as their workings and outcomes, which can be seen in figure 3 [2].

In essence, nearest neighbor graphs serve as an application-specific graphical notation, highly useful for nearest neighbor algorithms and clustering. However, the uniqueness of this notation makes it otherwise unsuitable for any other ML solution.

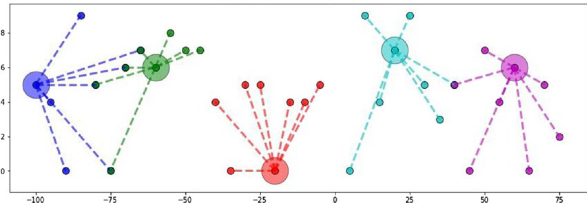


Figure 3: Nearest Neighbor Graph. Taken From [2]

3.4 Flow Charts

Flow charts are one of the most widely used graphical notations in ML solutions due to their effectiveness in showing interactions and the detailed workings of each step or component in the architecture. They can also include detailed images of what they represent, enhancing the understanding of the ML solution.

An advantage of flow charts is that they scale well when showing complex ML solutions. This advantage can be achieved using different graphical tools like matrices and examples, which help improve the flow chart by making it more compact, readable and easy to understand for the reader. Furthermore, flow charts are still very readable even with factual data, interactions and correlations that the ML solution shows [16] [31].

Flow charts have an advantage over automata, decision trees, and nearest neighbor graphs as they can be applied to most machine learning problems and solutions, including Supervised, Reinforcement, and Unsupervised Learning [29].

In reinforcement learning, flow charts are often used as it is possible to show the iterative structure that is needed for these ML solutions, e.g. figure 4. The repetition required in Reinforcement Learning can be shown through circles of activities and arrows. These can also interact with matrices, images or other supportive graphical notations.[2]

Another advantage of flow charts is that they can be divided and joined easily at any point. It is mostly possible due to their composition in various graphical notations. Consequently, the author can explain a subsection of the flow chart to help understand the main ML solution by taking a part of it and enhancing the detail of that subsection [14].

Flow Charts can also vary greatly in complexity for what is needed to be shown. The range goes from uncomplicated graphical notation like in the flow chart of a Reinforcement Learning solution in figure 4 to a fairly complex flow chart in figure 7 where BERTScore is illustrated. This ML solution is more complex and utilizes various graphical notations as support, such as heatmaps, matrices, arrows, activities, images, and examples. These notations help the reader better understand the complex ML method.

In conclusion, authors have a wide range of different complexities of flow charts to choose from to fit the ML solution they present. As such, flow charts offer a powerful means to visualize ML solutions and provide clarity and comprehensibility to readers.[16]

3.4.1 Activity

Activities in flow charts refer to bubble, circle or square shapes. These building blocks are connected through lines and arrows, often containing textual information, like data or ML solution features, to convey relevant information.

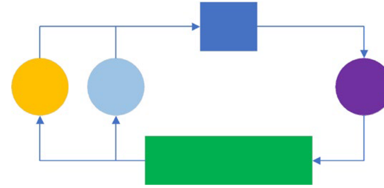


Figure 4: Flow Chart of a Reinforcement Learning Solution. Taken From [2]

They are also used to give examples through text, showing how the ML solution operates and evolves the example throughout the chart [16]. Images can also be integrated with activities by transitioning between text and visual representations [35].

Using these activities can give the reader a better understanding of the solution, as it gives an idea of how it works and enables the reader to easily follow the ML solution presented in the paper [16] [2].

3.4.2 Images

Images in flow charts show the connection between the real-world aspect and the ML solution. They can be easily integrated into a flow chart through arrows and texts to explain their relevance and how they relate to the ML solution or graphical notation.

These images play a vital role in enhancing the reader's understanding by showing the practical application of ML solutions in the real world. They can be photographs taken of real-world instances or custom-designed graphics. The designed graphics are often combined with another graphic notation to provide additional context, as depicted in figure 5. Notably, recent papers have emphasised providing images to reinforce the connection between the ML solution and its real-world aspect.[35, 16].

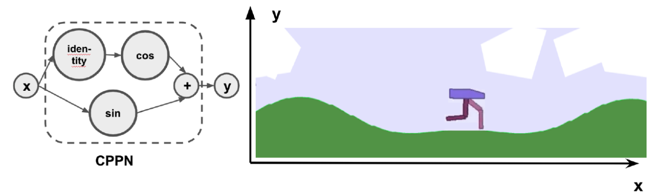


Figure 5: Example of an Image and a Flow Chart Interacting and Complementing Each Other. Taken From [35]

3.4.3 Heatmaps in Flow Charts

Heatmaps, another graphical notation we analyzed, are a supportive rather than a main tool for ML solutions. Their primary use lies in illustrating correlations and data, strengthening other graphical notations in the process, which as an example, can be seen in figure 6. Heatmaps are particularly valuable in showing the learning aspect of ML solutions within the context of a flow chart. In flow charts, they are

similarly used to matrices, as in figure 7.

An advantage of Heatmaps is their flexibility to show varying amounts of data depending on what is needed for the ML solution or to enhance the graphical notation [16] [22].

3.4.4 Matrices

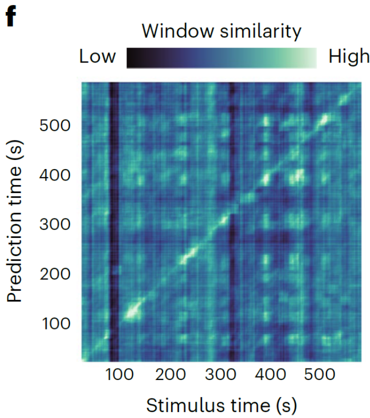


Figure 6: Example of a Heatmap. Taken From [16]

Matrices play a significant role in ML architecture, particularly in representing the interactions of nodes and states through transition matrices[31]. They are also utilized in a supporting role to represent how the data interacts within a model.

In this supporting role, they are often integrated into flow charts to enhance the reader's understanding of the utilized data, as shown in figure 7. While numbers are the most common representation within matrices, they can also incorporate text or colours to convey information.

While matrices do not represent a useful graphical notation, their importance lies in their ability to support other graphical notations. Their role as a supporting notation for flow charts and similar graphical notations is significant in clarifying and effectively conveying information [16].

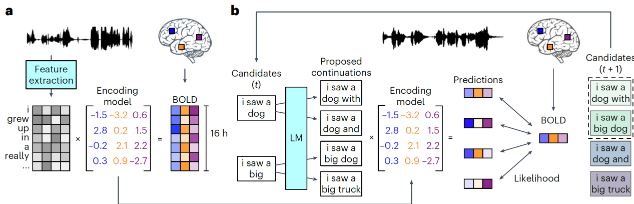


Figure 7: Example of Flow Chart With Matrices. Taken From [16]

3.5 Neural Network Diagrams

Despite using flow charts for NN architectures, more graphical notations are only related to the context of NNs. They are usually applied for classification and generative problems [1], which the data reflects. Despite their regular use, they are challenging to convey [39]. Among other things, this is also the reason why it is analyzed in more detail below.

3.5.1 Components

Usually, well-known networks are abstracted to components of a model visualized as boxes with a name label [17] [19] [11]. Sometimes more specific network details are depicted in the box [34]. Some common network types in which such visualization can be applied are a Recurrent Neural Network (RNN), Convolutional Neural Network (CNN) and Generative Adversarial Network (GAN) [37].

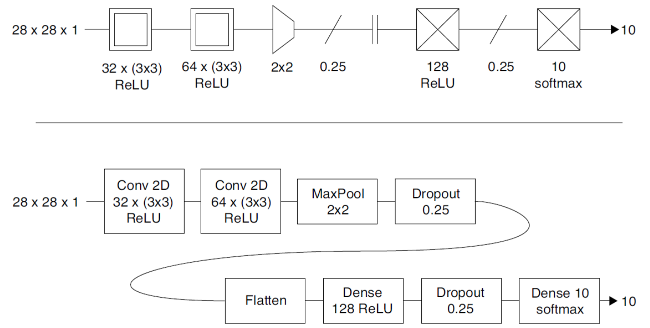


Figure 8: Top: Glassner's Schematic Form. Bottom: Traditional Box-And-Label Form. Taken From [1]

Fully Connected Networks Another typical network in our data is a Fully Connected Network (FNN) or Deep Network (DN). There exist multiple alternatives to visualize it. Using a base graph structure with continued dots is one way to do it [27] [43]. Another way is to abstract it as a normal component of the architecture with the name label "MLP" as an abbreviation for Multilayer Perceptrons [1] with some key characteristics like the width and length [11] [17] [20]. This terminology seems to be outdated and should not be used further. Instead, it should be labelled with the name "Dense" as in figure 8 and in figure 11 or "FNN" [1]. A simple graphical notation is a square with a cross, as depicted in figure 8. The cross intuitively conveys that each node between two layers is connected in pairs.

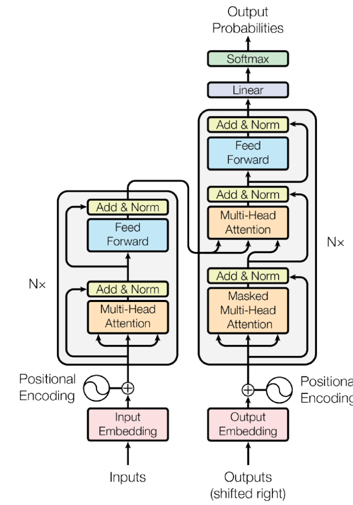


Figure 9: Transformer Encoder and Decoder. Taken From [4]

Encoder And Decoder Encoder and decoder, in their traditional sense, represent the operations to compress and decompress data like in autoencoder [37]. That is why they are often depicted as trapezes [27], indicating that the resulting data has a lower or bigger dimension.

For generators in GANs, [34], the trapeze visualization is also useful to represent an output of different dimensionality.

In transformer, RNN and other related architectures, encoder and decoder networks are very frequently part of the architecture. Therefore they are usually extra highlighted, like in figure with grey boxes 9 (see also for other architectures [42] [18] [24].

Since encoders and decoders are not always built the same, the sublayers of those networks are usually defined either in the same figure or in a separate figure. These sublayers are often networks themselves. Like in figure 9, both compo-

nents contain a feed-forward network.

Other more complex, self-defined components are typically further explained in another graphical figure or at the side, like in figure 10, which depends on the available space.

3.5.2 *Legends*

A legend is a perfect tool to increase the comprehensibility of used graphical components in architecture. For example, when the meaning of a component is not derivable from the figure caption, like in the above schematic representation in figure 8, a legend would make it clear. Four papers from our data collection included legends since they introduced multiple new meta-models. They had either no name label or had a label with an uncommon abbreviation [36], [19] [43] [10].

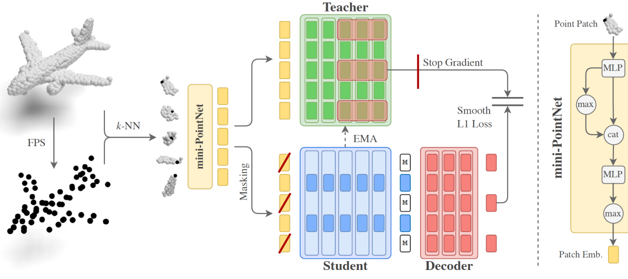


Figure 10: Input Data, Masked Data and Further Explained Components. Taken From [20]

3.5.3 *Layers*

Sometimes the architecture of a ML-Solution is a new category of an Artificial Neural Network [25] or a neural network of an existing kind but with some tweaks or adjustments. They often come with a certain advantage, such as shorter training times or a higher outcome quality. That is why there is a high interest in visualizing and highlighting them. Usually, these adjustments are made in the form of different configurations of various layers.

If the type of NN is defined, then each neuron in a layer generally does the same. Therefore, only layer-relevant characteristics are often stated. For example, in convolutional neural networks (CNN), each layer is usually depicted with a text label, including width, length, number of channels and activation function [12] [19] [21] [5] [3] [23]. In cases where some of this information is clear from the context, it can be left out.

Based on our data, layers are represented as nodes in the NN diagram and are labelled differently depending on the type of layer. Using a unique graphic to represent them is even better, as demonstrated in figure 8.

3.5.4 *Data Representation*

Several NN diagrams include images or visualization of their input data, as depicted in figure 10 with point clouds [15]. Multiple papers [30] [5] [43] [8] [26] [25] also included those visualizations during data flow like in figure 7 with sentences or in figure 11 with images. By following this approach, the architectures can be understood much faster and are more appealing to the eye.

Feature Visualization In certain research papers [25] [23] [3] [26] [40], the data produced by convolutional layers are

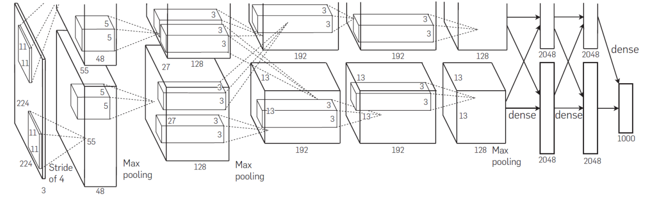


Figure 11: Data-Oriented Visualization of a CNN-Architecture. Taken From [23]

often depicted rather than the layers themselves. This data-oriented visualization is practical because convolutional layers and CNNs are commonly used in image-processing applications.

The image data, also called feature [1], is stored in a matrix and passed from one convolutional layer to another with typical techniques. Techniques include pooling, striding, filter size and the resulting data size. In figure 11, the cuboids are the features of a convolutional layer, which have the matrix size in (length x width x depth or channel) attached as information.

The smaller cuboids in the data cuboids are an example of a filter. Like above, the size of the filter is specified as (length x width). The quantity of channels is the same as for the feature. That is why the filter cuboid and feature cuboid have the same depth. The pyramid form with dashed lines on each filter cuboid visualizes the information that the neuron's output is combined into a (1x1xn) output matrix, which is part of the next feature cuboid.

More specific information about what the convolutional layer does with the input data can be specified between two cuboids, like in the same figure for the common operation "Max pooling".

Data Embeddings Usually, data also get abstracted to a collection of smaller data embeddings, which represent the data going through the diagram [8] [43] [4]. Figure 10 shows how data embeddings can be handled during data flow and support imparting information. The student is only fed with non-masked data embedding, conveyed with a red-slanted solid line for each masked data embedding. The results are then added to the earlier masked data and given to the decoder. Instead of using a concatenation or add sign, the input has the same data dimension with just the masked data parts signed with an "M". This approach is recommended since it mediates the process intuitively and space-efficiently.

3.5.5 *Table Visualization*

Another option to depict CNNs, including their layers, is in the form of tables, which convey key information about a network in a space-efficient manner [12] [21]. In our data, this graphical notation is only used for common or simple networks where the data flows from one layer to the other without alternative routes, unlike the architecture in figure 11. The advantage is that the architecture is more comparable to other NN architectures.

3.5.6 *Common Basic Operations*

Some layers or components have similar or the same functionalities and are used across different types of architectures. We define those components as basic operations.

Concatenation A very common basic operation is concatenation, which is usually stated as a mathematical add sign like in figure 9. A name label also suffices, but it should be distinct from other layers or components of the architecture. In the same figure, specifically in the encoder and decoder part, a fundamental component includes concatenation in the form of "add" in combination with another basic operation called "norm", whose functionality can also be derived from the name. The abbreviation "cat" (or "concat" [42]) is also valid, which was used in figure 10.

Pooling Pooling or downsampling is a technique to reduce the dimension of a matrix, which can be the input or output data of a whole network layer. The advantages include insensitivity to changes, memory needs, and execution time. That is why it is often used in different NN architectures, especially with convolution. Our collection of papers contains different visualization approaches for pooling. One of those is in the form of a triangle with a name label [15]. This visualization reflects the functionality of reducing the dimension very well and is unambiguous. Another similar alternative is a trapeze [1], but it should only be used if the context includes no encoder or decoder due to the likelihood of confusion. In such cases where the space for visualization is sparse, a good alternative is to use a small thin oval with a colour label combined with a legend [36]. Our data also includes negative examples. An oval or a circle representing a pooling layer gives, despite the contrast to other parts of the architecture, no information about what happens with the data during the process [20].

Dropout Another basic operation is dropout, which unlike before, relates more to the training of a NN. In the architecture, graphical notations for dropout layers specify how many neurons in the previous layer should be considered during training [1]. In figure 8, the notation includes a slanted solid line over the connecting line of those layers.

Activation Functions A standard for NNs is activation functions, which prevent the network from "collapsing into the equivalent of a single neuron", as Glassner states. (p.329) [1] Therefore, activation functions are very often implicitly assumed for each neuron. A better approach than implicitly assuming the type of function from the context is to use a name label for the type under the layer, like in figure 8 or in other papers [26] [3]. Some papers in our collection also included a mathematical graph for those functions, which, of all the options, was the best due to the intuitive understanding and non-ambiguity of other components in the architecture. Those graphics can be divided into two categories. Either way, the graphic gives the information when a function is linear or non-linear, or the information that approximates the outline of the graph has [8] [1]. In the first case, linear graphs are depicted with a straight line and non-linear graphs with a sinus curve [1] [13].

Another common activation function is softmax, whose visual representation is, as our data shows, sometimes handled differently than the others [28]. In our data, there has never been a graphic for this operation with the outline of a function graph. Only a box with a name label, like in figure 8 or in another paper [42]. These softmax boxes almost always occur at the end of the architecture, as we can infer from our research data.

Mathematical Functions And Signs If functions are used outside the context of activation functions [4] [8] [13] [6], then graphics depicting their outlines can also be used. That should only be used when graphics for activation functions are marked as such, for example, with the name label for the type.

Some architectures also included mathematical signs to add details that could not be easily expressed with graphical notations [18] [13]. This notation should be used as little as possible due to the rapidly increasing complexity it causes.

3.5.7 Neural Network Units

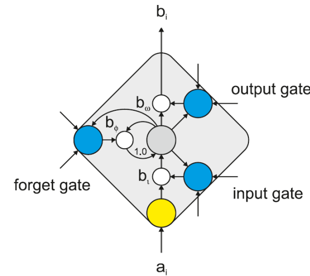


Figure 12: The Inner Structure of a NN Unit. Taken From [32]

In cases notable architectural changes were made in NN layers, it offers a graphical explanation of those changes. As our data reflects, a layer is visualized as a conventional flow chart [42].

Our data also includes close-up views from single neurons [32] [13]. Their overall structure resembles a conventional graph, but with the addition of incoming and outgoing arrows, as visualized in figure 12. As usual in graphs, nodes are depicted as circles with arrows directing the data flow from one node to the next. The neuron is visualized as a square object to differentiate it from the nodes.

4. DISCUSSIONS

For large and complex architectures, flow charts are the preferred graphical notation, as our data reflects, which is no surprise. Flow charts can be used in many ways and can include multiple supporting notations like heatmaps, images as data, matrices and special symbols.

During data exploration, we came across various papers which used little to no graphical notation [13] or just specified the mathematical models [41] to show their ML solution. This makes it harder for readers to apprehend the inner workings of the architecture, especially for readers with no computer science or mathematical background. In the latter case, we suggest the use of easily digestible visualizations. For example, they should include fewer mathematical formulas and good data visualization. That can be achieved with various notations stated in section 3.4.2, section 3.5.4 and section 3.5.5. Even if the visualization is not mathematically formally sufficient, it should be included for overview purposes of the core idea.

More known architecture components should be visually abstracted or shorted if the audience consists only of scientific researchers with ML backgrounds. As a result of the abstractions and shortenings, the notable changes in an architecture solution are emphasized. Examples of this common approach are FNN, CNNs, GANs, convolution layers or common basic operations, stated in section 3.5.5. This approach can also make sense if an architecture is complex and large.

Since an architecture's components might get unrecognisable at first glance, the usage of a legend would compensate

for it. This goes well with unique representations for different types of components, as it usually happens with basic operations and data embeddings in NN diagrams. Likewise, complex as well as newly introduced components can and should be explained in a separate, more detailed visualization as in Figure 10.

In cases where data representations are included in the visualization, which we also recommend for computer vision and natural language processing, a good ratio between content and data should be emphasized.

In the near future graphical notations like using data representations, legends and more abstractions are likely to increase in frequency due to the AI hype caused by ChatGPT and similar applications. It leads to more interest from a younger audience and non-graduates. An indicator of that change is that many universities have recently added ML and artificial intelligence to their curriculum of undergraduate programs. Furthermore, many research journals will probably start to see a rise in non-graduate readers. Consequently, researchers will have to design their papers carefully to address those audiences. Those visual abstractions of known components like encoder, decoder, NN types, and basic operations are slowly becoming more standardized.

After analyzing around ten papers from the last four years, we also came upon the trend that architecture visualizations will go away from complex hard to understand graphical notations. We also think that the trend will go more towards clear and easy-to-read graphical notations like flow charts and NNs as it improves the reader's understanding immensely, so the reader can comprehend the ML solution before reading the text.

5. CONCLUSIONS

Our analysis of around 50 papers allowed us to gather many graphical notations used over the past 34 years for ML architecture solutions. It would have been more effective to apply more selection criteria during data exploration for better coverage of used meta-models in the ML domain knowledge. Since it cannot be ruled out that other notations exist, we can only report partial success in pursuing our goal (**G1**).

Regarding our second objective (**G2**), we recommend graphical notations that are easy to read and enhance the understanding of the ML solution. We recommend using NN diagrams and flow charts with supporting graphical notations discussed in section 4.

With our third goal (**G3**) in mind, we tried in section 4 carefully to give a trend for graphical notations. This was particularly difficult considering how little it is known of future requirements of ML architectures and its fast-changing research landscape. Knowing we predict the trend for graphical notations will go more towards flow charts and standardizations of common components in an architecture. We also think that new, improved graphical notations will be found, leading graphical notations in ML solutions to new heights.

6. ACKNOWLEDGMENTS

We want to thank our Mentor Selin Aydin for her guidance and input.

7. REFERENCES

[1] A. Glassner. *Deep Learning: A Visual Approach.* 2021.

[2] A .Vermeulen. *Industrial Machine Learning: Using Artificial Intelligence as a Transformational Disruptor.* 2019.

[3] Alec Radford and Luke Metz and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2016.

[4] Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin. Attention Is All You Need, 2017.

[5] C. Elich, F. Engelmann, J. Schult, T. Kontogianni and B. Leibe. 3D-BEVIS: Birds-Eye-View Instance Segmentation. 2019.

[6] Carpenter, G.A. and Grossberg, S. and Markuzon, N. and Reynolds, J.H. and Rosen, D.B. Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3(5):698–713, 1992.

[7] Center für Künstliche Intelligenz. Machine Learning. `https://www.ai.rwth-aachen.de/cms/KI/Forschung/AI-Methods/~mbxkr/Machine-Learning/`. Accessed: 2023-07-06.

[8] Clinton Ansun Mo and Kun Hu and Chengjiang Long and Zhiyong Wang. Continuous Intermediate Token Learning with Implicit Motion Manifold for Keyframe Based Motion Interpolation, 2023.

[9] D. Walker and F. Myrick. Grounded theory: An exploration of process and procedure. *Qualitative health research*, 16:547–559, 2006.

[10] Fjodor van Veen. The Neural Network Zoo. `https://www.asimovinstitute.org/neural-network-zoo/`. Accessed: 2023-07-07.

[11] Francis Engelmann and Theodora Kontogianni and Jonas Schult and Bastian Leibe. Know What Your Neighbors Do: 3D Semantic Segmentation of Point Clouds. pages 395–409. 2019.

[12] He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian. Deep Residual Learning for Image Recognition. pages 770–778, 2016.

[13] Hochreiter, Sepp and Schmidhuber, Jürgen. Long Short-Term Memory. *Neural Computation*, 9:1735–1780, 1997.

[14] I. Sárándi, A. Hermansand B. Leibe. Learning 3D Human Pose Estimation from Dozens of Datasets using a Geometry-Aware Autoencoder to Bridge Between Skeleton Formats. 2023.

[15] J. Groß, A. Ošep and B. Leibe. AlignNet-3D: Fast Point Cloud Registration of Partially Observed Objects. 2019.

[16] J. Tang,A. LeBel, S. Jain and A. Huth. Semantic reconstruction of continuous language from non-invasive brain recordings. *Nature Neuroscience*, pages 1–9, 2023.

[17] Jiaqing Xie and Rex Ying. Fea2Fea: Exploring Structural Feature Correlations via Graph Neural Networks. pages 238–257. 2021.

[18] Jie Zhou and Ying Cao and Xuguang Wang and Peng Li and Wei Xu. Deep Recurrent Models with Fast-Forward Connections for Neural Machine Translation, 2016.

[19] Jingxia Jiang and Tian Ye and Jinbin Bai and Sixiang Chen and Wenhao Chai and Shi Jun and Yun Liu and Erkang Chen. Five A^+ Network: You Only Need 9K Parameters for Underwater Image Enhancement, 2023.

[20] K. Abou Zeid, J. Schult, A. Hermans and B. Leibe. Point2Vec for Self-Supervised Representation Learning on Point Clouds. 2023.

[21] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015.

[22] Kassambara, Alboukadel. *Practical guide to cluster analysis in R: Unsupervised machine learning*. 2017.

[23] Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, 2017.

[24] Kyunghyun Cho and Bart van Merrienboer and Caglar Gulcehre and Dzmitry Bahdanau and Fethi Bougares and Holger Schwenk and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014.

[25] LeCun, Y. and Boser, B. and Denker, J. S. and Henderson, D. and Howard, R. E. and Hubbard, W. and Jackel, L. D. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989.

[26] LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey. Deep learning. *Nature*, 521:436–444, 2015.

[27] Lemley, Joe and Bazrafkan, Shabab and Corcoran, Peter. Deep Learning for Consumer Devices and Services: Pushing the limits for machine learning, artificial intelligence, and computer vision. *IEEE Consumer Electronics Magazine*, 6(2):48–56, 2017.

[28] Levi, Gil and Hassncer, Tal. Age and gender classification using convolutional neural networks. pages 34–42, 2015.

[29] Liu, Yanli and Wang, Yourong and Zhang, Jian. New machine learning algorithm: Random forest. pages 246–252, 2012.

[30] M. Knoche, I. Sárándi and B. Leibe. Reposing Humans by Warping 3D Features. 2020.

[31] M. Nilsson and M. Ejnarsson, Marcus. Speech recognition using hidden markov model, 2002.

[32] M. Sundermeyer, R. Schlüter and H. Ney. LSTM neural networks for language modeling. pages 194–197, 2012.

[33] Nahar, Nadia and Zhang, Haoran and Lewis, Grace and Zhou, Shurui and Kästner, Christian. A Meta-Summary of Challenges in Building Products with ML Components–Collecting Experiences from 4758+ Practitioners. 2023.

[34] P. Achlioptas, O. Diamanti, I. Mitliagkas and L. Guibas. Learning Representations and Generative Models for 3D Point Clouds, 2018.

[35] R. Wang, J. Lehman, A. Rawal, J. Zhi, Y, Li, J. Clune and K. Stanley. Enhanced poet: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. pages 9940–9951, 2020.

[36] Samiul Based Shuvo and Syed Samiul Alam and Syeda Umme Ayman and Arbil Chakma and Prabal Datta Barua and U Rajendra Acharya. NRC-Net: Automated noise robust cardio net for detecting valvular cardiac diseases using optimum transformation method with heart sound signals, 2023.

[37] Sarker, Iqbal H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. *SN Computer Science*, 2(6):420, Aug 2021.

[38] Schumacher, Tobias and Wolf, Hinrikus and Ritzert, Martin and Lemmerich, Florian and Grohe, Martin and Strohmaier, Markus. The effects of randomness on the stability of node embeddings. pages 197–215, 2022.

[39] Seifert, Christin and Aamir, Aisha and Balagopalan, Aparna and Jain, Dhruv and Sharma, Abhinav and Grottel, Sebastian and Gumhold, Stefan. *Visualizations of Deep Neural Networks in Computer Vision: A Survey*, pages 123–144. 2017.

[40] Taigman, Yaniv and Yang, Ming and Ranzato, Marc'Aurelio and Wolf, Lior. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. page 1701–1708, 2014.

[41] Troy Maasland and João Pereira and Diogo Bastos and Marcus de Goffau and Max Nieuwdorp and Aeilko H. Zwinderman and Evgeni Levin. Interpretable Models via Pairwise permutations algorithm. *CoRR*, abs/2111.09145, 2021.

[42] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016.

[43] Zhijun Zhai and Jianhui Zhao and Chengjiang Long and Wenju Xu and Shuangjiang He and Huijuan Zhao. Feature Representation Learning with Adaptive Displacement Generation and Transformer Fusion for Micro-Expression Recognition, 2023.

Comparing Selected Security Modeling Languages Using SEQUAL Framework

Haydar Genc
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
haydar.genc@rwth-aachen.de

Mohamed Amine Zormati
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
mohamed.amine.zormati@rwth-aachen.de

ABSTRACT

Nowadays, safeguarding valuable assets has become a paramount concern for companies. However, it is impractical to address every security threat that arises. To deal with these challenges, *Security Modelling Languages* (SMLs) have been developed to encompass Security Risk Management.
Nonetheless, selecting the most suitable language can be a daunting task, as it requires comparing and evaluating these languages to determine the most fitting solution. In this paper, we present initially an overview of a certain group of SMLs together with their respective metamodel. In addition, we make use of SEQUAL assessment framework to compare the language assessments based on their syntax, semantics, and pragmatics. At the end, we evaluate the comparative analysis and come to the conclusion that there is no best SML to cover all the security threats that might appear under certain circumstances. The choice of a SML always depends on the security aspects one wants to model.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*software configuration management, SMLs, SecureUML, Mal-Activity-Diagrams, PrivUML, SEQUAL framework*

1. INTRODUCTION

In today's world, software applications are connected to the internet and are facing major risks of potential hazards from data leaks to the failure of critical secure systems, which may be fatal and costly. For this reason, software security plays a vital role in the system development process.

Modeling solutions at an architectural level becomes essential for developing effective security measures. Popular languages like UML provide the necessary elements to express and document the required building blocks. The modeling language choice in security context is unclear.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2023/24 RWTH Aachen University, Germany.

The contribution of this paper lies in its in-depth comparison of some prominent SMLs, by comparing their metamodels based on the SEQUAL Framework [5]. It is within the scope of this paper to identify and highlight the characteristics and properties of each language. Through an empirical research and literature review, it provides a comprehensive analysis that aids developers in selecting the appropriate language for their specific security requirements.

In conclusion, this paper will address the following research questions (RQs):
RQ 1: How can SMLs be compared with each other?
RQ 2: How do the selected SMLs SecureUML, MADs, and PrivUML compare?

2. DEFINITIONS

In this section, we define the terminology used in this paper, because these terms are defined differently by various authors.

Security modelling languages (SMLs) are defined languages that allow the representation, analysis, and reasoning regarding various aspects pertaining to computer networks, systems, and software applications. They serve by providing an organized framework for communicating critical ideas surrounding topics such as policies, threats, and vulnerabilities alongside potential countermeasures.

SMLs typically consist of a set of well-defined syntax and semantics, along with a collection of constructs and rules that allow for the representation of security properties, system components, their interactions, and the security controls in place. Depending on their needs they may deploy graphical representations or written documentation (sometimes even combining both approaches) enabling analysts and engineers alike an opportunity to model their systems' security-properties in a precise and formal manner.

Metamodel: In the context of modeling languages, a metamodel refers to an abstract representation or description of a language itself. It defines the structure, concepts, relationships, and rules that govern the construction and interpretation of models within that particular modeling language. In this paper, a metamodel provides a formal representation for defining the syntax and semantics of the modeling language, allowing users to create consistent and meaningful security models. It defines the types of elements or entities that can be used in the models, such as assets, threats, vulnerabilities, controls, and relationships between them. It specifies associated properties and attributes of these elements.[2].

According to Faily et al. [11], the metamodel of a SML serves as a foundation for creating specific models within that specific language. It establishes the vocabulary and grammar that users can utilize to describe security-related aspects, such as risk assessments, access control policies, system architectures, and attack scenarios. By adhering to the metamodel, users can ensure that their models are conforming to a standardized representation and can be understood and processed by tools or other stakeholders.

3. SELECTED SMLS

During our literature review we found several modeling languages that claim to be SMLs like SI*[16], UMLSec [13], SecureUML [22], Mis-Use-Case [1], SecureTropos [19], Knowledge Acquisition in Automated Specification (KAOS) [17], PrivUML [18], UMLintr [12], Mal-Activity-Diagrams (MADs) [23], SecureBPMN [6] and PaML [8].

For our comparison, we are aiming for SMLs which have an existing graphical representation of their metamodels which is not too complex. Some modeling languages have metamodels that are too big and hard to compare with other metamodels, thus are exceeding the scope of this work. Hence, we selected the following SMLs: SecureUML, MADs and PrivUML. In this paper, we'll introduce and compare these languages in terms of metamodel expressiveness and quality properties. We will present characteristic elements of PrivUML and SecureUML to demarcate these SMLs from UML. Since MADs are not based on UML, we will not provide example usages MADs' elements.

3.1 SecureUML

SecureUML is designed for incorporating essential access control information into UML-based models by providing a set of new notations. These notations encompass various aspects of access control such as roles, permissions associated with roles (role-permission), and assignments of users to roles (user-role assignment). With its comprehensive access control model, extensibility, visual representation, and the capability to create designs at a high level of abstraction, SecureUML is highly suitable for designing secure systems, conducting business analysis, and generating design models for diverse technology stacks [22]. Developers without extensive security expertise can utilize SecureUML to build secure systems by implementing the defined model [4].

SecureUML builds upon the foundation of Role-Based Access Control (RBAC) and extends it by endowing it an ability of authorization constraints. RBAC is an approach that restricts system access to authorized users by assigning roles and thus permissions to users. An authorization constraint sets conditions for accessing a specific action [15].

Creating A SecureUML Model

In order to create a SecureUML model, there is, according to Andrey et al[22], a workflow of 6 steps that must be followed. To make these steps concrete, we are going to apply them to the example of an online banking system.

Identify Users: In the online banking system context, Alice assumes the client role and is represented as a class labeled with the stereotype "<<user>>" in the system.

Identify application roles: Within the online banking system, a role called "account holder" is defined to enable the client to access the system's services. This role is represented as a class with the stereotype "<< *r*ole>>".

Map users into roles: In the system, the user named Alice is linked with the role of "account holder".

Identify resources: In this example, the bank account is considered a resource and is depicted as a class labeled with the stereotype "<<ModelElement>>".

Identify actions: The stereotype "permission" is utilized as an association class with the <<permission>>" stereotype to establish actions within the system. By employing the attribute "action type", it defines various actions. For instance, in our example, The user Alice can view her account through the system and also update it. That's why, "read" and "update" can be considered as actions.

Identify authorization constraints: In order for an authorization to be applicable, certain authorization constraints must be fulfilled. These constraints can be viewed as preconditions. For instance, in this example, the constraint *BusinessHoursOnly* ensures that transactions can only be performed during business hours.

Account for cardinality: SecureUML also includes the representation of cardinality, which illustrates the relationship between Account holder and Bank account as a one-to-many relationship. The relationship between the User Alice and Account hold is a one-to-one relationship.

Metamodel Of SecureUML

SecureUML metamodel (shown in figure 1) extends the UML metamodel by introducing three new concepts: User, Role, and Permission.

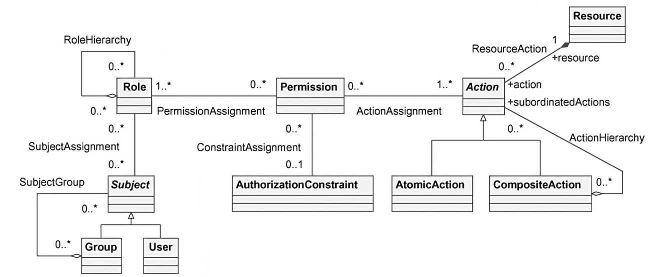


Figure 1: SecureUML metamodel (adapted from [4])

The "Subject" type serves as the base for all users and groups in the system, where it is an abstract type that cannot be directly instantiated. Users represent system entities, while groups name sets of users and groups. Subjects are assigned to groups through the "SubjectGroup" aggregation, which establishes an ordering relation. Subjects are assigned roles via the "SubjectAssignment" association.

Roles represent jobs and encompass all the privileges necessary for carrying out those jobs. Permissions grant roles access to one or more actions. The actions are assigned through the "ActionAssignment" association, and the entitled roles are indicated by the "PermissionAssignment" association. Permissions must be assigned to at least one role and action due to cardinality constraints. Roles can be hierarchically ordered through the "RoleHierarchy" aggregation, with the aggregate role inheriting all the privileges.

Authorization Constraints are logical predicates attached to permissions through the "ConstraintAssignment" association. These constraints make the validity of permissions dependent on the system state, such as the current time or attribute values. The constraints are expressed using OCL

(Object Constraint Language [24]) expressions, where the system model defines the vocabulary and includes the additional symbol "caller", representing the user on whose behalf an action is performed.

The "Resource" class serves as the base for all model elements representing protected resources in the system modeling language. Actions that can be performed on these resources are represented by the "Action" class. Each resource offers one or more actions, and each action belongs to a single resource, indicated by the "ResourceAction" composite aggregation. Actions are categorized into atomic actions, which directly map to target platform actions, and composite actions, which are high-level actions used for grouping purposes. Composite actions form an "ActionHierarchy."

3.2 Mal-Activity-Diagrams

Mal(icious)-Activity-Diagrams build upon the principles of UML Activity diagrams, focusing specifically on the behavioral aspects of security issues [23]. Constructing a MAD involves creating a regular process and then incorporating undesired behavior into it. [7].

MADs enhance UML activity diagrams by introducing a negative element to depict security and dependability concerns within the process. These diagrams use the same syntax and semantics as UML activity diagrams but add extra syntax elements to enhance their functionality: [7], [9]

Malicious activities represented using regular activity icons, but with their colors inverted. **Malicious actors** portrayed as misusers, distinguished by white text on a black background. These malicious actors, also known as misusers, are assigned dedicated swimlanes. **Malicious decision boxes** represented using regular decision boxes, but with a black filling.

Metamodel Of MADs

MADs begin with an InitialState as the starting point and conclude with a FinalState as the endpoint. Within the diagram, there are three types of activities: Activity, MalActivity, and MitigationActivity. All these constructs are encompassed within an AnySwimlane, which can be either a Swimlane or a Mal-swimlane. A Swimlane contains SwimlaneElements, which can be an Activity, a MitigationActivity, or a Decision.

Activities in the diagram specify a parameterized sequence of behavior. MitigationActivities represent process improvements aimed at preventing MaliciousActivities. Decisions illustrate branching based on the order of rejected or accepted conditions.

A Mal-swimlane consists of Mal-swimlaneElements, which can be Mal-activities or Mal-decisions. Alongside Mal-activities, a Mal-swimlane may also include legitimate activities. Mal-activities are performed by malicious actors with the intention to disrupt the normal process. In some cases, a legitimate user may unknowingly perform a Mal-activity when deceived by an attacker. Mal-decisions refer to decisions made with malicious intent.

3.3 PrivUML

PrivUML is a SML that aims to provide its user with a model-driven design-approach that allows the design of secure and privacy-friendly systems. Thus, UML is being used as foundation for a more profound SML that allows to model consolidated access control information that guarantees privacy in its design. In the following paragraphs, we will define the terms: personal information and privacy.

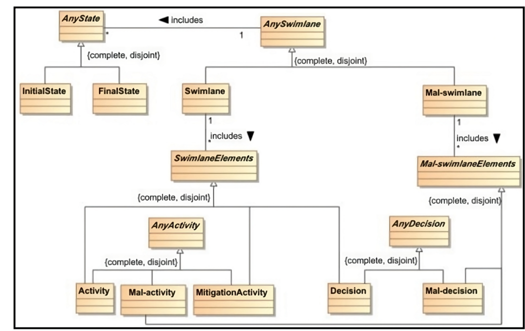


Figure 2: Metamodel of MADs [7]

According to [20], personal information takes in scope any factual/objective or subjective information, whether recorded or not, about an individual that can be identified.

Privacy can be defined formally in four definitions: usage purpose of the data, access right visibility (who is allowed to see the data), the level of detail of the information and the amount of time the data is stored [3].

Fundamental Of PrivUML: OrBAC Model

Since PrivUML is an extension of OrBAC we want to take a look at the OrBAC model. OrBAC is an access control model that stands for organization-based access control. The central component of an OrBAC model is the organization. Privileges are part of a role that gets assigned to a subject. Thus, privileges do not apply directly to the subjects. Permissions are expressed through the predicate "Authorization (organization, role, activity, view, context)" [18]. RBAC and OrBAC are both models for access control, but their focus is different. So, according to the OrBAC model, a role can be permitted to perform a certain activity with an organization's view on the activity in a context. Context is used to constrain the access rules. In OrBAC a privilege does not solely include permission. It can optionally include a prohibition and an obligation independently of each other. To cover privacy protection requirements, the data owner's consent is needed to access his data/for a subject to perform an activity on owner's data. This can be modeled with the OrBAC context [18].

PrivUML is able to model: the OrBAC model, terms of access (permission, ban, and obligation), consent of the data owner, access objectives and a hierarchy of object views.

PrivUML Metamodel

PrivUML expresses the hierarchy at the level of the entities "View" and Role. The data owner can define the same view with of objects with different levels of detail, thus two subjects are allowed to access the same object with different levels of detail. The PrivUML metamodel uses the entities "RSO, "AO" and "VOO".

RSO (Role of the Subject in the Organization): An association class that allows a subject to play multiple roles in an organization [18]. As an example, an employee that plays the role of a "service worker" at online bank A can play the role "customer" at online bank B.

AO (Activity in the Organization): An association class, AO, links activities and organizations, allowing the meta-

model to structure the same activity in different ways. [18].

As an example, we are linking the activity of transferring money at a online bank. Online bank A transfers money by reducing the sent amount in the senders' database entry and adding the sent amount in the receiver's database entry. Online bank B instead transfers money by withdrawing the sent amount of money from the sender's bank account and deposits it to the receiver's bank account. The activity "transferring money" is linked with two different organizations, and thus the activity is structured according to the needs of the different online banks.

VOO (Object View in Organization): An association class that links object views with organizations. Objects are associated with VOOs to define the same object view differently [18]. For example, the view "money command" corresponds in online Bank A to database tables. In online bank B "money command" is defined as a money safe.

The association class *Access Modality* links the entities "RSO", "AO", "VOO" and "Context" to express the permission, the prohibition, or the obligation, provided in a context to a role to perform activities on views in an organization [18]. Action is an association class that links the entities "AO" and "VOO". An action depends on the activity and the view of objects in the organization. In case of two supermarkets A and B, the activity "Delete" in organization A is the action "delete", while the activity "delete" corresponds to the action "remove". "A subject with a role allowing an action on an object can perform this action if the consent context is positive." [18]. Consent can be established by the object's owner and for a specific object/in favor of the subject requesting access/according to the purpose of access/to communicate the explicit permission of the object's owner expressed as an OCL constraint [18].

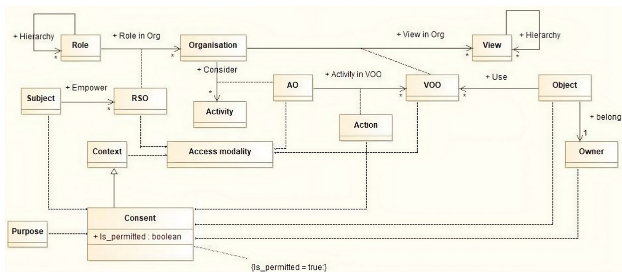


Figure 3: PrivUML meta-model [18]

4. ASSESSING THE SMLS

4.1 The SEQUAL Framework

The SEQUAL Framework is a reference model for assessing the quality of models. It recognizes three distinct types of qualities: semantics, syntax and pragmatics [21], [5]. As a result, it establishes essential principles for evaluating model quality. However, it maintains an abstract nature, and it is dedicated to general-purpose models and not only security models [14]. Consequently, we will introduce a set of measures to comprehend the quality of security models.

Semantic quality refers to the alignment between a model and its semantic domain. We evaluate semantic quality using the following qualitative properties and measures:

-*Semantic completeness:* This property assesses whether the model encompasses all the necessary functionalities required by the software. In the context of security modeling, a language should incorporate concepts that pertain to the Role-Based Access Control (RBAC) domain. We selected RBAC since it is often used in identity and access management (IAM). To measure semantic completeness, we calculate the ratio between the number of RBAC concepts represented in the model and the total number of RBAC concepts (a total of 7: Users, Roles, Operations, Objects, Permissions, Permission assignment, User assignment).

-*Semantic correctness:* This property examines whether the model accurately represents the required developmental aspects. In the security domain, it necessitates distinguishing between data-related concerns and security-related concerns, since the security model should solely incorporate security-related knowledge. The qualitative measure for this property is the "Percentage of security-related statements" in the model.

-*Executability* refers to the presence of a technology that can take the model as input and effectively translate it into a functional implementation. The determining factor here is the availability of such technology. This characteristic is gauged using the measure "Technology capable of executing the model", which indicates whether the necessary technology exists to carry out the implementation process.

Syntactic quality pertains to ensuring syntactic correctness, meaning that the statements within the model adhere to the syntax rules and relevant conventions of the modeling language. The following qualitative properties and their corresponding measures are established: -*Syntactic validity:* This property evaluates whether the grammatical expressions used in the SML are correct. The measure assigned to assess this qualitative property is the "Number of syntactically invalid statements". A higher value indicates poorer syntactic validity in the SML.

-*Syntactic completeness:* This property examines whether all the grammar constructs and their constituent parts are present in the SML. To assess syntactic completeness, we use the measure "Number of incomplete statements".

Pragmatic quality addresses the alignment between the actors' interpretation of the model and their existing domain knowledge. Consequently, the emphasis lies on how participants perceive and interpret the model. To evaluate this aspect, we establish the following qualitative properties and their respective measures:

-*Annotation:* This property focuses on whether a reader can easily identify elements that are likely to undergo changes. This is particularly significant as new system security policies are frequently introduced. Being aware of relevant sections and efficiently implementing new security concerns can reduce system maintenance costs. The measure "Number of annotation elements" indicates the count of annotations used in the model.

-*Modifiability:* This property measures the ease with which the structure and content of the model can be modified. This aspect is especially crucial in the security model, given that system security policies may require frequent changes. To estimate modifiability, we employ the measure "Time spent to modify" (in minutes), which indicates the duration required to alter the security policy within the system.

-*Cross-referencing:* This property examines whether the various components of the model content are interconnected. To assess cross-referencing, we utilize the measure "Number

of cross-reference links", which quantifies the count of links established between different model components.

4.2 Results

In this section, we provide the outcomes of our evaluation of the SMLs. The results for the three quality types are summarized in Table 1. As specified in the previous section, we assessed semantic quality based on semantic completeness, semantic correctness, and executability. Syntactic quality is represented by measures of syntactic validity and syntactic completeness. Lastly, pragmatic quality is defined in terms of annotation, modifiability, and cross-referencing.

Semantic quality			
Qualitative property	**SecureUML**	**MAD**	**PrivUML**
Semantic completeness	100%	57.14%	42.85%
Semantic correctness	100%	50%	100%
Executability	YES	NO	YES
Syntactic Quality			
Qualitative property	**SecureUML**	**MAD**	**PrivUML**
Syntactic validity	1	0	1
Syntactic completeness	0	0	0
Pragmatic quality			
Qualitative property	**SecureUML**	**MAD**	**PrivUML**
Annotation	3	0	5
Modifiability	5–10	2-5	2-5
Cross-referencing	3	1	3

Table 1: Evaluation of the SMLs

5. COMPARISON OF THE SMLS

In this section, we cover the comparison on SMLs based on our SEQUAL Framework assessment.

5.1 SecureUML And MADs

Our evaluation revealed that SecureUML and MAD exhibit equal scores only for syntactic completeness. However, two qualitative properties were identified as being stronger in MAD compared to SecureUML(i.e., syntactic validity and modifiability). On the other hand, the remaining five qualitative properties were found to have higher evaluations in SecureUML when compared to MAD. In terms of semantic completeness, SecureUML strives to capture a comprehensive set of security-related functionalities, including those pertaining to RBAC. It ideally incorporates all seven RBAC concepts.

MADs are primarily focused on modeling malicious activities and security threats, rather than comprehensive RBAC-based functionalities. As such, the expectation for semantic completeness in MADs may not directly align with the RBAC concepts. However, MADs should still include relevant security-related concepts specific to malicious activities and threat modeling.

When comparing SecureUML and MADs based on semantic correctness, SecureUML, being a more comprehensive security modeling language, exhibits a higher percentage of security-related statements, reflecting its focus on broader security aspects. On the other hand, MADs would emphasize security-related statements that pertain to modeling malicious activities specifically.

In terms of executability, SecureUML holds an advantage over MADs due to the likely availability of tools and technologies supporting SecureUML models. While SecureUML can be interpreted and translated into functional implementations using existing technology, the absence of dedicated technology for MADs limits their executability. This limitation can hinder the practicality and usefulness of MADs.

MADs score better in Syntactic Validity compared to SecureUML. This indicates that MADs exhibit a higher level of correctness in terms of the grammatical expressions used within their models. The "Number of syntactically invalid statements" is lower for MADs, suggesting a stronger adherence to syntax rules and conventions compared to SecureUML. Both SecureUML and MADs achieve the same score in terms of Syntactic Completeness. This implies that both languages encompass all the necessary grammar constructs and their constituent parts.

SecureUML outperforms MADs in the property of Annotation, which means SecureUML incorporates a higher number of annotation elements, indicating its ability to facilitate the identification of elements likely to undergo changes.By efficiently implementing new security concerns, SecureUML can contribute to reduce the overall maintenance costs of the system. On the other hand, MADs may have a lower number of annotation elements, potentially making them less effective in highlighting areas requiring modifications. Conversely, MADs surpass SecureUML in terms of Modifiability. MADs exhibit greater ease in modifying the structure and content of the model, as indicated by a shorter "Time spent to modify" measure. This indicates that MADs allow for more efficient alteration of the security policy within the system.

SecureUML has a higher number of cross-reference links, indicating a stronger inter-connectivity among the various components of the model content. This characteristic ensures that relationships between different model components are well-established and effectively captured. MADs, have a relatively lower count of cross-reference links, potentially indicating a lesser degree of interconnection among its model components.

5.2 SecureUML And PrivUML

SecureUML outperforms PrivUML in terms of the qualitative property of Semantic Completeness. However, five other qualitative properties, namely Semantic Correctness, Executability, Syntactic Validity, Syntactic Completeness, and Cross-referencing, are evaluated equally for both SecureUML and PrivUML. Nevertheless, PrivUML excels in the remaining two qualitative properties, namely Annotation and Modifiability.

PrivUML, being a modeling language tailored for privacy concerns, has a different emphasis compared to SecureUML in terms of its completeness in representing RBAC concepts. While RBAC might still play a role in PrivUML, it might not be the primary focus. Instead, PrivUML may prioritize privacy-related concepts, such as data sensitivity, access control policies, or privacy-enhancing technologies.

SecureUML and PrivUML have the same semantic correctness because both modeling languages aim to accurately represent the required developmental aspects, encompassing both security and privacy considerations. They focus on capturing the necessary knowledge, concepts, and rules related to security and privacy within their models.

PrivUML and SecureUML are both executable. Due to its broader adoption and longer history, SecureUML has a mature ecosystem of tools and technologies that can effectively execute the models (i.e., MagicDraw tool that, based on some rules, transforms the SecureUML-model into a code, which could be executed through Oracle database management system). This means that SecureUML models can be translated into functional implementations with greater ease and reliability. PrivUML, being a more specialized modeling language for privacy aspects, have a relatively smaller ecosystem of supporting technologies. While there are some technologies available for PrivUML, their availability and maturity might be more limited compared to SecureUML. Both SecureUML and PrivUML have the same score in Syntactic Validity. This indicates that both languages demonstrate an equal level of correctness in terms of the grammatical expressions used within their models.

Similarly, both SecureUML and PrivUML achieve the same score in terms of Syntactic Completeness. This implies that both languages incorporate all the necessary grammar constructs and their constituent parts to the same extent.

PrivUML outperforms SecureUML in the property of Annotation. PrivUML exhibits a higher number of annotation elements, indicating that it offers better support for identifying elements likely to undergo changes. This enables readers to efficiently implement new security concerns and reduce overall maintenance costs. On the other hand, SecureUML may have a relatively lower number of annotation elements, which could make it less effective in highlighting areas requiring modifications.

Similarly, PrivUML surpasses SecureUML in terms of Modifiability. PrivUML demonstrates greater ease in modifying the structure and content of the model. This means that implementing changes to the security policy within the system is more efficient in PrivUML. In contrast, SecureUML may require more time and effort for modifications, potentially hindering flexibility in adapting to frequent changes in system security policies.

Both SecureUML and PrivUML achieve an equal score in Cross-referencing. This means that both languages effectively establish links between different components of the model content.Therefore, in terms of Cross-referencing, both SecureUML and MADs exhibit an equal level of effectiveness in capturing the relationships between different elements.

5.3 MADs And PrivUML

Our comparison of MADs and PrivUML revealed that PrivUML and MADs only have equal score in two qualitative properties, syntactic completeness and Modifiability. MADs surpass PrivUML in regard to semantic completeness. MADs are inferior to PrivUML in any other qualitative properties (i.e., semantic correctness, syntactic validity, annotation and cross-referencing.

In regard to semantic completeness, MADs surpass PrivUML because MADs are more aligned with the RBAC concepts than PrivUML is. PrivUML is based on the OrBAC model and thus focuses on organization-based access-control which indeed overlaps with RBAC concepts (in terms of Roles, Operations, and Objects) but does not put its focus on RBAC concepts. On the other hand, MADs are focused on modeling malicious activities. Thus, MADs do not focus on RBAC concepts and only allow modeling four out of seven RBAC concepts. When comparing MADs with PrivUML in regard to semantic correctness, we found out that PrivUML can derivate solely security-related statements where MADs also allow non-security-related statements. In terms of executability, PrivUML likely has the tools needed to be implemented by an engine. PrivUML can be transformed to Transformation of PrivUML into XACML Using QVT (A model-to-model transformation language) [10]. After transforming PrivUML into XACML, we can make use of XACML based authorization engines (e.g., Balana) to implement the PrivUML model as code in a structured manner following the rules of XACML. MADs have an absence of technical tools, and thus MAD is not executable.

The amount of syntactically invalid statements is lower for MADs than for PrivUML. This relation between the number of syntactically invalid statements and the SMLs suggests that MADs have a stronger adherence to syntax rules than PrivUML. Thus, MADs have a better syntactical validity than PrivUML.

MADs and PrivUML have the same score regarding syntactical completeness. Both MADs and PrivUML achieve the same score in terms of Syntactic Completeness. This implies that both languages encompass all the necessary grammar constructs and their constituent parts.

Since the number of syntactically incomplete statements is equal for both languages, indicating that they meet the requirements for syntactic completeness equally well. PrivUML outperforms MADs in regard to annotation, which means that PrivUML incorporates a higher number of annotation elements. This fact indicated that PrivUML makes it easy to identify elements which are likely to undergo changes. MADs may have a lower number of annotation elements, potentially making it less effective in highlighting areas requiring modifications.

PrivUML and MADs score equally in the qualitative property modifiability. Altering a security policy takes around 2–5 minutes for both SMLs.

PrivUML has a higher number of cross-reference links than MADs. This means that the relationships between different model components in PrivUML are well-established and effectively captured. On the other hand, MADs have a significantly lower count of cross-reference links, which indicates a lower degree of interconnection along its model components.

6. DISCUSSION

You do not need classifications for SMLs in order to compare them with each other. The SEQUAL Framework is a reference model for assessing the quality of models. To perform a SEQUAL assessment on SMLs, you need the metamodel of the SMLs.

When we sum up our comparison from sections 4 and 5, we realized that MADs performed the worst. SecureUML and PrivUML score equally in the qualitative property syntactic quality. Since SecureUML outperforms PrivUML in semantic quality and PrivUML outperforms SecureUML in pragmatic quality, we cannot conclude a “best” SML. Regarding the SEQUAL assessment, you can pick SecureUML as a SML, if semantic quality is more important for you than pragmatic quality. If it is the other way around, we would suggest PrivUML.

These conclusions are based on the results of an assessment on SMLs with our adaption of the SEQUAL Framework, which does not consider the focus of a SML. For exam-

ple, the SEQUAL Framework assesses SMLs based on their RBAC implementation and not all SMLs focus on RBAC. PrivUML for example takes in account OrBAC, MADs focus on modeling malicious activities and others take a look at the sociotechnical aspects of security. Thus, the level of implementation of RBAC concepts may not be the best criteria to compare non RBAC SMLs. When choosing a SML to work with, make sure to check if you need a specialized SML. If so, take a SML that focuses on the area that has to be modeled. Otherwise, you can choose a more general SML. It always depends on the security aspects the security engineer focuses on.

Thus, we question if the SEQUAL Framework is a good comparison method for SMLs. The qualitative properties of the SEQUAL Framework had to be adapted to work with SMLs. Besides that, we found it difficult to compare SMLs with different foundational concepts like RBAC and OrBAC with each other. We propose the idea of comparing SMLs with the same security focus and adapting the qualitative property semantic completeness according to the foundational security concepts category of SMLs tries to model. For example, for OrBAC modeling languages, we would adapt semantic completeness to the ratio of the amount of modelable OrBAC concepts in one SML divided by the amount of concepts needed to model OrBAC optimally. The reference number here would be the number of different concepts from OrBAC. The numerator would be the amount of OrBAC concepts implemented by the SML. SMLs that try to model different foundational security concepts can still be compared with each other, but then the comparison of semantic completeness would not be as decisive as comparing two SMLs that try to model the same foundational security concepts with each other because it is trivial that a SML from one area e.g., SecureUML (has a focus on RBAC) is better at modelling RBAC concepts than a SML with focus on OrBAC concepts e.g., PrivUML.

The only case where such a comparison makes sense is when looking for a SML that can model multiple foundational security concepts. When looking for a SML that can model OrBAC and RBAC, we suggest taking RBAC and OrBAC based SMLs and assess them based to the SEQUAL Framework with an adapted semantic completeness. SMLs can be compared with each other by utilizing the SEQUAL Framework. We presented the idea of assessing a SML on multiple different security paradigms to find a SML that can model two or more foundational security paradigms. The only issue was that the results were not summed up yet. We propose an equation to calculate the semantic completeness of an SML that is being assessed on multiple foundational security paradigms. A security paradigm in this case is, e.g., OrBAC. Let x be the semantic completeness, l_1 the number of concepts applied by the SML l from the modeling paradigm d_1 and d_{1_a} the amount of security concepts from the modeling paradigm $d1$. Thus, l_2 the number of concepts applied by the SML l from the modeling paradigm $d2$ and d_{2_a} the amount of security concepts from the modeling paradigm $d2$. All other variables are defined analogously. Equation for semantic completeness:

$$x = \frac{\frac{l_1}{d_{1_a}} + \frac{l_2}{d_{2_a}} + ... + \frac{l_n}{d_{n_a}}}{|\{d_1, d_2, ..., d_n\}|}, \text{ where } n \in \mathbb{N} \text{ fixed but arbitrary}$$

Two SMLs are equivalent to each other if and only if their metamodels are fully transformable into each other. Two languages that are SMLER equivalent have the same expressiveness and thus are semantically equivalent. We propose the SML-Equivalence-Relation (SMLER): Let L be the set of all SMLs and T be the set of all model-to-model transformation languages. $(l_1 \equiv l_2) \equiv l_1 \sim_{SML} l_2 \equiv \exists t_1, t_2 \in T : t_1(l_1) = l_2 \wedge t_2(l_2) = l_1$, where $l_1, l_2 \in L$ The SMLER is transitive, reflexive, and symmetrical. These properties make the relation an equivalence relation. This property is useful when forming SML-Equivalence-Classes (SMLECs).

Now we can derive statements about the equivalence of SMLs. We can extend the concept of SML equivalence by introducing SMLECs. These equivalence classes allow us to categorize equivalent SMLs and sort out redundant SMLs, which helps when choosing between SMLs. Since each SMLEC focuses on different security paradigms, we can assess SMLs from one SMLEC with our adaption of the SEQUAL Framework and then choose the SML that fits our need regarding all qualitative properties (without semantic completeness). $[l_1]_{\sim SML} = \{l_2 \in L | l_1 \sim_{SML} l_2 \wedge l_1, l_2 \in L\}$ The expenses of comparing multiple SMLs with each other was at $\binom{n}{k}$, where n is the amount of SMLs to compare with each other and $k = 2$ is the amount of SMLs you can compare directly with each other. Thus, the upper limit of comparisons of SMLs when comparing their SMLECs is in worst-case $\binom{n}{k}$ comparisons. Let T be the set of all model-to-model-transformation-languages, L be the set of all SMLs and t(x) be defined as the function that executes the model-to-model-transformation language t on the SML x with the resulting SML on the right side of the equation. $t : L \rightarrow L, t(l_1) \rightarrow l_2$,where $l_1, l_2 \in$ L.

Comparing SMLs from different SMLECs does not make sense because they focus on different security paradigms.

7. CONCLUSION

This research paper emphasizes the significance of software security in today's world. By comparing prominent SMLs using the SEQUAL Framework, the study provides developers with valuable insights for selecting the most suitable language based on their specific security requirements. The analysis focuses on the meaningfulness and quality properties of the metamodels of SecureUML, Malicious-Activity Diagrams (MADs), and PrivUML. The research questions posed in this pre-study set the stage for further discussions and exploration in the field of SMLs. Overall, this paper serves as a valuable resource for enhancing software security practices and aiding decision-making processes in software development. The findings and insights presented in this paper will inform future studies and contribute to the advancement of software security practices in an increasingly interconnected world.

8. FUTURE WORK

We propose the idea of a table that categorizes all SMLs into their equivalence classes regarding the SMLER and assesses the resulting SMLEC based on our adaption of the SEQUAL Framework with semantic correctness according to the security modeling paradigm of the equivalence class, which may be more than just one paradigm. The fact that the SMLER is transitive is very helpful here.

We propose a decision framework that supports the modeler in choosing the adequate SML. The idea is to categorize

SMLs by security modeling paradigms to give an overview over all SMLs and their general capabilities. We think that it is likely that all SMLs can be abstracted to their core functionalities, which are represented by the security modeling paradigms. Additionally, an adaption of the SEQUAL Framework could be used to assess these security modeling paradigms on a higher level.

We propose introducing a set of scales to sum up the qualitative properties of the SEQUAL Framework to display semantic quality and the other metrics by one single number per quality metric. These 3 metrics can be summed up further to give the SEQUAL Framework an ability of assessing a SML by solely a single number. Now, the expenses to compare SMLs with each other are significantly easier than having to compare each qualitative property of the SEQUAL Framework of a language with the SEQUAL assessment of another language. SMLs are now comparable by one metric. We call this metric the "SEQUAL Score".

9. REFERENCES

[1] I. Alexander. Misuse cases: use cases with hostile intent. *IEEE Software*, 20(1):58–66, Jan. 2003.

[2] U. Aßmann, S. Zschaler, and G. Wagner. Ontologies, meta-models, and the model-driven paradigm. *Ontologies for software engineering and software technology*, pages 249–273, 2006.

[3] K. Barker, M. Askari, M. Banerjee, K. Ghazinour, B. Mackas, M. Majedi, S. Pun, and A. Williams. A data privacy taxonomy. In *Dataspace: The Final Frontier*, pages 42–54. Springer Berlin Heidelberg, 2009.

[4] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):39–91, 2006.

[5] C. Cares and J. Franch Gutiérrez. Towards a framework for improving goal-oriented requirement models quality. In *12th Workshop on Requirements Engineering*, pages 3–14, 2009.

[6] Y. Cherdantseva, J. Hilton, and O. Rana. Towards SecureBPMN - aligning BPMN with the information assurance and security domain. In *Lecture Notes in Business Information Processing*, pages 107–115. Springer Berlin Heidelberg, 2012.

[7] M. J. M. Chowdhury. Towards security risk-oriented mal activity diagram. *International Journal of Computer Applications*, 56(10), 2012.

[8] P. Colombo and E. Ferrari. Towards a modeling and analysis framework for privacy-aware systems. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*. IEEE, Sept. 2012.

[9] M. El-Attar. From misuse cases to mal-activity diagrams: bridging the gap between functional security analysis and design. *Software & Systems Modeling*, 13:173–190, 2014.

[10] J. El Mokhtari, A. Abou El Kalam, S. Benhaddou, and J.-P. Leroy. Transformation of privuml into xacml using qvt. In *Proceedings of the 12th International Conference on Soft Computing and Pattern Recognition (SoCPaR 2020) 12*, pages 984–996. Springer, 2021.

[11] S. Faily and I. Fléchais. A meta-model for usable secure requirements engineering. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, page 29–35, New York, NY, USA, 2010. Association for Computing Machinery.

[12] M. Hussein and M. Zulkernine. UMLintr: a UML profile for specifying intrusions. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*. IEEE, 2006.

[13] J. Jürjens. UMLsec: Extending UML for secure systems development. In *≪UML≫ 2002 — The Unified Modeling Language*, pages 412–425. Springer Berlin Heidelberg, 2002.

[14] J. Krogstie. SEQUAL as a framework for understanding and assessing quality of models and modeling languages. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 1611–1620. IGI Global, July 2014.

[15] L. Lúcio, Q. Zhang, P. H. Nguyen, M. Amrani, J. Klein, H. Vangheluwe, and Y. L. Traon. Advances in model-driven security. In *Advances in Computers*, pages 103–152. Elsevier, 2014.

[16] F. Massacci, J. Mylopoulos, and N. Zannone. Security requirements engineering: The SI∗ modeling language and the secure tropos methodology. In *Advances in Intelligent Information Systems*, pages 147–174. Springer Berlin Heidelberg, 2010.

[17] m. mbd. Developing secure and safe systems with knowledge acquisition for automated specification (kaos). 2021.

[18] J. E. MOKHTARI, A. A. E. KALAM, S. BENHADOU, and H. MEDROUMI. PrivUML: A privacy metamodel. *Procedia Computer Science*, 151:53–60, 2019.

[19] H. MOURATIDIS and P. GIORGINI. SECURE TROPOS: A SECURITY-ORIENTED EXTENSION OF THE TROPOS METHODOLOGY. *International Journal of Software Engineering and Knowledge Engineering*, 17(02):285–309, Apr. 2007.

[20] T. Piper. The personal information protection and electronic documents act-a lost opportunity to democratize canada's technological society. *Dalhousie LJ*, 23:253, 2000.

[21] K. Rekstad and J. Krogstie. Using sequal for identifying requirements to ecore editors. *arXiv preprint arXiv:2202.02565*, 2022.

[22] A. Sergeev. Role based access control as secureuml model in web applications development with spring security. *Ph. D. dissertation, Master thesis*, 2016.

[23] G. Sindre. Mal-activity diagrams for capturing attacks on business processes. In *Requirements Engineering: Foundation for Software Quality: 13th International Working Conference, REFSQ 2007, Trondheim, Norway, June 11-12, 2007. Proceedings 13*, pages 355–366. Springer, 2007.

[24] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., USA, 2 edition, 2003.

Classification of modernization methods based on experience reports: A Systematic Literature Review

Leila Mangonaux
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
leila.mangonaux@rwth-aachen.de

Marco Heinisch
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
marco.heinisch@rwth-aachen.de

ABSTRACT

Modernization of legacy systems is considered a complex task, beginning with the selection of the method to be used. Numerous and diverse strategies exist, and each comes with its own set of advantages and disadvantages. Due to a lack of comprehensive classification of these strategies and their characteristics, the evidence-based choice of such a strategy for a given legacy system is challenging. In this work, we performed a systematic literature review of experience reports with the aim of providing guidance for future software modernization processes. We propose five major characteristics of legacy systems and types of migrations upon which suitable modernization methods can be selected. They serve as general guideline which takes into account the goals and available resources.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

Keywords

legacy system, modernization, migration, systematic literature review, SLR

1. INTRODUCTION

A legacy system [33] is commonly understood as an old system that still provides value and remains in operation within an organization. Such systems often provide core business functionalities and are critical. Due to this importance the functionality of these systems must be preserved.

However, maintainability costs are high and flexibility or extensibility of such systems is difficult due to old or lacking design principles and outdated technology. In addition, there is often a lack of knowledge as employees with expertise leave or retire over time and documentation is incomplete or outdated. Thus, dealing with legacy systems is challenging.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2023 RWTH Aachen University, Germany.

To fulfill new requirements and to reduce maintainability costs, modernization is needed [1]. As this is a task prone to failure, best practices, experience, and recommendations are needed, to be able to decide which approach is the most appropriate one with the highest success chance for a given legacy system. However, due to a broad variety of methods for software modernization and the lack of a standard approach, the identification of a suitable strategy is complex.

Case studies and experience reports about past modernization procedures are available, but no review has been performed yet. Our main research question is to find out, how software modernization methods can be categorized, by conducting a systematic literature review (SLR). We aim to find evidence-based characteristics of both the applied strategies and modernized software. This paper's contribution is a classification of the applied methods by these characteristics to provide an indication for future selections of modernization methods.

Our paper is structured as follows: In section 2 we discuss related work and SLRs. We then give an overview of definitions in section 3, before explaining how we conducted the SLR in section 4. In section 5, we report our results and propose our categorization of methods. We discuss these results in section 6 and finally give a conclusion in section 7.

2. RELATED WORK

Jamshidi et al. [22] performed a systematic literature review on cloud migration. They analyzed 23 publications and characterized four types of cloud migration. They found that partial migration and cloudification were the most common types, followed by migrations of the whole stack. But they did not analyze why the reviewed cases were migrated in the chosen manner, and therefore cannot provide guidance for future modernizations.

A second SLR in the field of software modernization was performed by Khadka et al. (2013) [25]. They analyzed 121 publications on SOA (service-oriented architecture) migration and found that the selection of a specific architecture is the most common method for target system understanding. The authors also conclude that candidate services are mostly identified manually and wrapping is the most common technique for SOA implementation. However, this SLR lacks an analysis of the reasons for the migrations.

Althani and Khaddaj [1] give an overview of migration methods by defining a categorization in their systematic review. They identified three major categories: complete, incremental and partial migration. A complete migration, also known as big bang, describes a complete redevelopment of

the system from scratch. Incremental migration signifies a gradual re-implementation while partial migration describes the migration of parts of the legacy system. However, the categories are defined solely by the migration approach, instead of considering the characteristics of the legacy system.

Another categorization is described by Comella-Dorda et al. [10]. They use the term of *software evolution* and group this in three subclasses: maintenance describes small changes aiming to keep the system running, modernization encompasses more extensive interventions and replacement entails a redevelopment of the entire system. The authors subclassify modernization in white box modernization, which encompasses reverse engineering before reengineering, and black box modernization. Comella-Dorda et al. discuss approaches, including wrapping and GUI reengineering, for black box modernization only.

Khadka et al. (2014) [24] conducted interviews with professionals on the topic of software modernization. Thus, they identified which benefits of legacy systems the interviewees perceive: this includes the business logic encompassed in the code, the reliability and performance that were reinforced through use and extensive tests. As drivers towards modernization, the authors identify the inflexibility with regard to changes in the system, rising maintenance costs, the lack of knowledge in the form of documentation and expertise and the risk of system failure. Though these findings coincide with our results, they don't provide guidance on the type of modernization to apply for a given legacy system.

3. BACKGROUND

In the course of our work, we have encountered a variety of terms and concepts used and defined differently by various authors. In this section, we therefore define the most important concepts and have referred to these definitions for the remaining work.

3.1 Legacy system

The definition of *legacy system* proposed by Stamati et al. [33] picks up on many aspects, such as the large efforts put into it, typical for this kind of software: *"It describes an old system that remains in operation within an organisation and often represents a massive, long-term business investment that could be composed of extremely efficient robust and fine tuned applications built over many years by a combination of IT and business experts."*

However, in this work, we consider a broader variety of legacy systems, including software for individual products, rather than solely software designed for internal use of companies. These systems can include a lot more than the code itself, as indicated by Dedeke [12] who defines *"a legacy system as an aggregate package of software and hardware solutions whose languages, standards, codes, and technologies belong to a prior generation or era of innovation"*.

Indeed many of the reviewed systems strive towards more modern technologies. Yet, we consider a legacy system independent of the age or era of the previously used technologies but rather define it as a *"conjunction of soft- and hardware, code, documentation and technology that is in use although it does not fulfill the current requirements set by the organization running it"*.

3.2 Reengineering

Chikofsky and Cross [8] define reengineering as follows: *"Reengineering (...) is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system."*

Further, we will clarify the newly introduced concepts of *forward engineering* and *reverse engineering*. Reverse engineering describes the process of analyzing code with the aim of gaining full understanding by identifying its components. Forward engineering, in turn, encompasses the reimplementation of the system (often based on the results of the reverse engineering process), mostly according to a defined architecture or paradigm.

3.3 6R's: six migration strategies

Based on the descriptions by Orban [2] and Hayretci and Aydemir [20] of cloud migration strategies, we have defined the following six migration strategies which we will refer to as the 6R's:

1. **Rehost**: migrating the legacy system to a new platform without any changes to the code
2. **Replatform**: migrating the legacy system to a new platform with minimal adaptions (e.g. wrapping)
3. **Repurchase**: replacing the legacy system with a commercial product
4. **Refactor/Rearchitect**: migrating the legacy system to a new architecture, programming language or paradigm with extensive changes to the code
5. **Retire**: shutting down the legacy system without replacing it
6. **Retain**: no migration or modernization of the legacy system

All in all, our redefinition of the 6R's aims to make them applicable to any type of migration, rather than restricting it to cloud migration, as does the original definition.

3.4 Restructure and Refactor

Chikofsky and Cross [8] also provide a definition of the term *restructuring* as *"the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics)."*

In the reviewed papers, restructuring was more commonly referred to as *refactoring*. As Mens et al. state in their study on refactoring [28] that the definition of the terms is "basically the same", we will use them interchangeably.

3.5 Refactor and Rearchitect

While refactoring and rearchitecting are terms often used interchangeably in relation to the 6R's, we have found that these modernization strategies describe different methods when used in other contexts. We therefore distinguish between the two terms, defining refactoring as above and referring to other changes in the system architecture, programming language or paradigm as rearchitecting. However, we do consider both approaches as a single category in the 6R's

as they are often used together and the extent of the changes made to the legacy system is comparable.

4. RESEARCH METHOD

To identify relevant case studies and reports, we conducted a Scientific Literature Review as proposed by Kitchenham [26]. This approach reduces the risk of bias and allows reproducibility. Having identified the need for a review in section 1, we completed the initial step of the SLR. The further steps are explained in the following sections.

4.1 Research Question

The primary goal of the SLR is to identify categories of modernization method characteristics based on experience reports. More precisely, we formulate the following research questions:

> **RQ 1**: How can software modernization methods be categorized?
> **RQ 1.1**: What are the characteristics of legacy systems that influence the choice of the modernization method?
> **RQ 1.2**: What are the characteristics of the chosen modernization methods that are relevant for a decision?

4.2 Search strategy

4.2.1 Search string

To find relevant literature addressing our research questions, the search string must identify papers that contain *experience reports* on *modernization* efforts related to a *legacy* system. Additionally, we considered synonyms to collect as much relevant material as possible. After conducting initial test searches, we developed the following search string:

> Title:(”case study” OR ”case studies” OR ”report”) **AND** Abstract:(”modernisation” OR ”modernization” OR ”re-engineering” OR ”reengineering” OR ”migration”) **AND** Abstract:(”legacy”)

We limit results to include *(”case study” OR ”case studies” OR ”report”)* in the title instead of in the abstract, as we had too large numbers of false positives and non-case studies when testing previous search strings.

4.2.2 Sources

Kitchenham [26] listed relevant databases a comprehensive software engineering SLR should consider. Due to limited search and filtering possibilities, we excluded *SpringerLink*, *CiteSeerx*, *Wiley InterScience* and *Google Scholar* from these, as well as *El Compendex* due to lack of access.

For each remaining database, we adapted the search string to the respective syntax and search functionalities. We had to deviate from the stated search string in *dblp* due to limited search syntax by using *”case”* instead of *”case studies” OR ”case study”*. If possible, we used filtering functions to already apply the selection criteria we define in the following section. The number of resulting papers of the conducted searches on the databases are presented in table 1.

4.2.3 Selection Criteria

After conducting the automated searches, we imported the initial results into a Citavi project. In the next step, we verified the relevance of each result, as we encountered false positives such as ”Cadastral positioning accuracy improvement: A case study in Malaysia”. Therefore, we defined and applied the following selection criteria.

All criteria are meant to improve and ensure the quality of the results. Fist, we consider only literature in English language and with available full-text to assure the accessibility of the content. A final publication stage and specific document type (Article, Conference Paper, Book (Chapter), Poster) is required to ensure a high scientific standard of results. Additionally, we filter by subject area and only include papers attributed to related topics (Computer Science, Engineering, Decision Science, Business Management, Social Science). To fulfill our research goal, we only consider literature containing experience reports that describe the modernization of a legacy system.

Subsequently, we processed the initial results as follows: We first filtered the publications by reading the titles, then also considered the abstracts. For the remaining papers, we took the full text into account. Statistics of the selection process are provided in table 2.

Table 1: Search Results

Sources	Results	Accessed
ACM	14	01.05.2023
dblp	19	10.05.2023
IEEE	24	01.05.2023
Inspec	59	07.05.2023
Scopus	90	04.05.2023
Sum	206	

Table 2: Number of excluded papers by criteria

Criteria	Remaining papers
Initial	206
Duplicate	-98
No PDF	-8
by title	-19
by abstract	-30
by content	-10
Result	41

4.2.4 Data Extraction

For each result, we identified the modernization method(s) applied in the case study, as well as justifications for the chosen method or reasoning about other methods. We also collected reasoning stated in the paper, about other not applied methods. In Citavi, we grouped our findings based on the 6R’s and sorted justifications and methods by categories to gain a comprehensive initial overview of our findings.

Figure 1: Number of papers by publication year

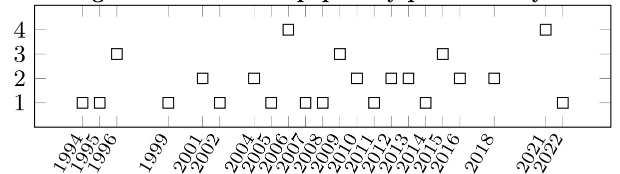


Figure 2: Number of applications [1] of a method by publication year

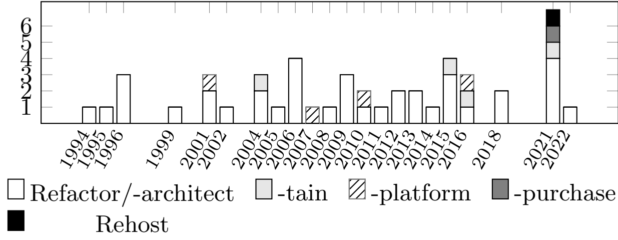


Figure 3: Split of applied methods in resulting papers

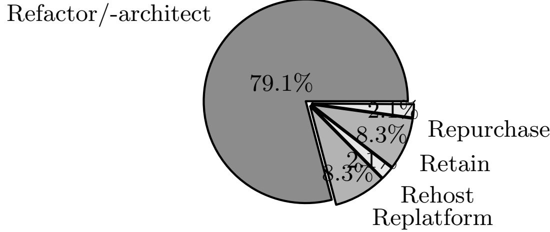
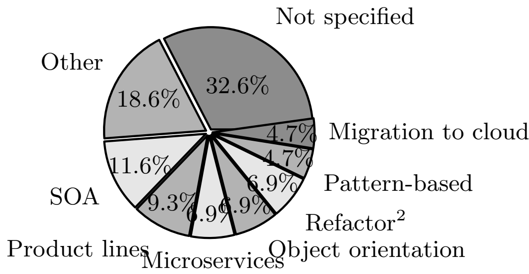


Figure 4: Split of applied submethods of "Refactoring/Rearchitecting"



5. RESULTS

5.1 Demographics

Figure 1 and 2 give an overview over the distribution of the reviewed studies. We cannot observe any significant trend in published experience reports on modernization procedures, implying a steady interest in the field. The same applies when considering the temporal distribution by strategy. Moreover, this visualization already indicates how predominantly *refactoring/rearchitecting* is selected over the other strategies. The proportional distribution is shown in figure 3 and confirms that 79.1% of the reviewed case studies applied this method. We therefore provide more detailed insight in the distribution of different refactoring/rearchitecting methods in figure 4, showing that the most popular architectures are the service-oriented and product line architectures. However, 51.2% of the target architectures were selected only once (other) or not further specified.

5.2 Findings on applied methods and their justifications

In the following section, we describe our findings, particularly the justifications for the chosen modernization methods mentioned in the reviewed publications.

Big bang and iterative approach

There are two general approaches that can be considered for software modernization: the replacement of the entire system or application at once, often referred to as 'big bang', or an iterative approach which modernizes the system step by step [30]. Although mentioned in the reviewed publications, the big bang approach was never selected as modernization approach. This was mostly due to the risks and costs involved which could be mitigated using an iterative or incremental approach. Table 3 lists the advantages for the respective approaches mentioned in the reviewed publications.

Table 3: Justifications for big bang (b) and iterative (i) approaches

b	avoids co-existence of two potentially incompatible systems	[34]
b	cheaper than iterative approach if this requires extensive reverse engineering	[11]
i	lower risks/higher success chances (e.g. with regard to limited time/experience)	[15], [34]
i	lower costs	[6]
i	customers see modernization efforts from the beginning	[34]

6R's

After the selection of a suitable modernization approach, the specific strategy must be chosen. We have classified these by the 6R's defined in section 3.3 which define migration strategies. All of the reviewed modernization procedures can be viewed as a type of migration (towards a new architecture, programming paradigm, language etc.), thus these categories can be applied. The tables 4 to 9 summarize the justifications for and against the individual strategies, indicated with + and -.

5.2.1 Retain

Retainment is used in four of the reviewed case studies, though in all cases it is used for parts of the legacy system only. While the rest of the systems undergo larger changes, such as rearchitecting, the retained systems are not in the scope of the main modernization effort. There are two major reasons mentioned for these decisions which are tightly connected to the characteristics of the respective legacy system.

Table 4: Justifications for (+) and against (-) retaining

+	preserves well functioning subsystem	[3], [31]
+	preserves rarely used/uncritical subsystem	[19], [20]

5.2.2 Rehost

Rehosting (also referred to as 'lift and shift' ([36])) was used in a single reviewed publication ([20]). In this case, the application was rehosted to a cloud platform and subsequently rearchitected.

[1]As some papers (partly) apply multiple methods, there is a difference between sums in figure 1 and figure 2
[2]See definition in 3.4

Table 5: Justifications for (+) and against (-) rehosting

+	fast migration	[20],[36]
+	minimal adaptations needed	[36]
-	not suitable if target system is not usable to full capacity without adaptation	[36]

5.2.3 Replatform

In the reviewed publications, replatforming, also referred to as 'soft migration' by Fürnweger et al. [3], was selected four times as modernization strategy. A typical method used is wrapping which implements an adapter around legacy code to integrate it in the new platform. It was applied to the server component of a system, when migration to web ([5]) as well as a data management system during database migration ([21]). Furthermore, half of the replatforming case studies describe the migration of databases, i.e. from a DB2 database to an Oracle database ([14]) and from a network database to a relational database ([21]).

Table 6: Justifications for (+) and against (-) replatforming

+	preserves critical data	[5]
+	minimal changes needed	[5], [21]
+	high feasibility if programming language of legacy system is well suited (e.g. Java is designed for platform independence)	[3]
-	no long-term solution for legacy systems of poor quality	[5]

5.2.4 Repurchase

Repurchasing proves to be a rarely described method in the publications we reviewed: it was selected in one, and mentioned as consideration or optional strategy for the future in two other reports.

Table 7: Justifications for (+) and against (-) repurchasing

+	suitable for applications with standard functionality	[20]
+	saves costs	[32]
-	risk of breach of copyright	[15]
-	not suitable for sensitive software	[15]
-	non-standard functions must be added	[32]

5.2.5 Retire

We could not find any description of software retirement in the publications we reviewed.

5.2.6 Refactor/Rearchitect

Refactoring was used in several publications. The main goals were improvement of the understandability and legibility of the code whilst preserving the semantics. This was especially important for legacy systems that were highly appreciated by the developers.

Table 8: Justifications for (+) and against (-) refactoring

+	minimal changes avoid functionality and performance loss	[17]
-	lower code quality if used without redesign	[29]

Rearchitecting was the most commonly used type of modernization. The major reason for rearchitecting was the aim for better maintainability, flexibility and availability. In addition, there is a trend towards adapting the legacy systems to modern technologies as well as the services and products they offer. We have deduced this from the most popular types of architectures selected and the reasoning given, listed in table 9.

Further system architectures and programming paradigms were described in the reviewed reports. As these were mentioned only once and/or lacking justifications, we cannot present findings on the reasoning for these strategies.

However, we have found that some legacy systems were described as leaning towards an architecture without fully implementing it. These systems were then modernized according to the identified architecture, using the predisposition it displayed. For example, Goedicke and Zdun [18] identified that the observed legacy system, though it implemented parts of multiple paradigms, was "almost structured according to the component-paradigm". They therefore chose to fully implement the component-orientation, aiming for more flexibility and understandability.

Table 9: Justifications for (+) and against (-) rearchitecting by architecture/paradigm

Service-oriented architecture (SOA)		
+	efficient delivery, addition and change of business processes	[16]
+	flexibility	[6], [16]
+	reuse of services and cost reduction	[6], [16], [20]
+	suitable for integration/merge of existing systems	[31], [6]
Product-line architecture		
+	enables faster product development	[7], [27]
+	higher flexibility in the applications	[23]
+	code reuse and cost reduction	[7], [23], [27]
Microservice-based architecture		
+	increased scalability	[20], [35]
+	resilience to changes	[20], [35]
+	improved code structure due to modularization	[4]
Object orientation		
+	increased reusability, maintainability, scalability and portability	[9]
+	higher level of abstraction	[9]
+	code reuse and cost reduction	[11]
-	may confuse developers who are unfamiliar with this paradigm	[15]
Pattern-based design		
+	increased understandability, reusability and maintainability	[9]
+	facilitated communication between developers	[9], [18]
+	allows integration of meta-programming techniques	[18]

5.3 Categorization

We have found that many of the justifications are related to similar characteristics of either the legacy system itself or the respective modernization strategy. Consequently, we have grouped these recurring factors into five dimensions.

Regarding *RQ 1.1*, we identify two main categories or dimensions of characteristics of a legacy system that influence the choice of a modernization method in the presented case studies:

1. Quality of the legacy system includes various aspects of quality and was already identified by Althani and Khaddaj [1]. A system with good implementation and predominantly consistent architecture, comprehensive documentation, or a disposition for change or modernization, for example to component-based architecture [18], enables different methods than a system not fulfilling these. Our results show that such systems can, for example, be replatformed or rehosted without large changes to the architecture whereas systems with low quality have to undergo restructuring or rearchitecting during migration to ensure better performance.

2. Extent of non-standard functionalities and business value of the legacy system is decisive for the choice between big bang/repurchasing and other methods and was already introduced as a second category by Althani and Khaddaj [1]. According to the publications we analyzed, the fewer non-standard functionalities are provided by the legacy system, the more suitable repurchasing is, since only few adjustments would have to be made.

Regarding *RQ 1.2*, we identify three categories of characteristics of the modernization methods themselves that are relevant for a decision:

1. Extent of adaption/modification. We perceive this as a continuous scale starting with *not changing anything* or *maintenance* followed by *modernization* and ending with *replacement*. It also allows structuring different classifications, such as the 6R's: Here, the method with no or little adaption is *Retain*, followed by *Rehost*, *Replatform* and *Refactor/Rearchitect*. Finally, the most comprehensive change is *Repurchasing* or *Retiring*. This category is highly related to the extent of new requirements and business needs that the legacy system cannot fulfill.

2. Speed or time-to-market. Time constraints caused by limited resources or urgency of the modernization greatly restrict the suitable modernization strategies. The less time is available for the procedure, the less extensive the applied strategy can be. For example, the big bang method was frequently rejected due to the effort it would have entailed, whereas iterative refactoring (without rearchitecting) and replatforming were chosen due to the little reverse engineering they require. Yet, the case studies showed that fast modernization methods are frequently short-term solutions which offer a lower quality system in comparison to other, more time-consuming, methods.

3. Risk or success chances vary widely from one legacy system to another and may be difficult to estimate. However, we have observed that the way the developers deal with risks and changes is mentioned in several case studies and can be assessed before and during the modernization by including them in the process. The less these developers are willing to adapt to the changes and take risks, the less invasive these changes should be in order to avoid overwhelming the people concerned. Such methods could, for example, be refactoring, thus preserving the semantics and minimizing risks.

Other categories such as cost and available personnel are relevant but rarely discussed in the identified literature and therefore not stated here.

To give an answer to *RQ1*, we suggest the categorization based on the dimensions stated above. Using these allow classifying strategies by factors that are relevant as guidelines determining whether or not to use them.

In the case of deciding which strategy for a modernization of a legacy system fits best, the stated dimensions need to be analyzed and prioritized for this specific task. For example, characteristics of the system (quality) and the current requirements (extent of adaption/modification) need to be carefully considered. Similarly to the concept of *The Golden Triangle*[3], not all priorities can be fulfilled for a specific case. But defined priorities and current state knowledge allow us to narrow down the options and focus on those modernization strategies that serve the prioritized goals.

6. DISCUSSION

6.1 Discussion of results

Though we were able to classify all methods used in the reviewed publications, the extraction of reasoning of the choice of method proved to be less obvious. We have observed that the justifications were often not explicitly stated. Instead, the anticipated benefits were highlighted without further justification why the selected method would achieve these best. Yet, this permitted an analysis of the major drivers towards a modernization and goals for the given software. We found these results to coincide well with the results presented by Khadka et al. (2014) [24] (see section 2).

A striking result is the lack of documented rehosting, retainment, retirement and big bang processes in the reviewed publications. These were not or rarely mentioned in the case studies and often rejected. Due to this gap, we can only suggest explanations for this phenomenon: we cannot distinguish, whether these gaps are due to rare use of the methods or rare documentation thereof.

The mentioned methods may rarely be applicable to given legacy systems. For example, rehosting may rarely be employable, because this would require changes to the system, making the modernization a replatforming process. Retiring and replacing legacy systems may not be considered due to the importance of many legacy systems. Many case studies mention the extent of business logic implemented in legacy systems and the reliability of such proven software which make many systems indispensable. Moreover, retiring systems may simply not be considered whilst they are functioning, even if they are superfluous.

The shortage of evidence on the use of these methods can also be caused by lacking reporting. We have noticed that the majority of case studies only describe the part of the legacy system that is rearchitected, making a differentiation between complete and partial migrations difficult. This implies that many other subsystems are retained or retired and these procedures are not considered worth a report. The same could apply to rehosting processes, as this is one of the least complex migration approaches.

During our research, we noted that some authors, such as Thiele [13], include *rewrite*, a complete re-implementation of the legacy system, as a migration strategy. Although we did not consider this initially, we later found that it was not selected in the reviewed case studies.

[3]https://beingaprojectmanager.com/nuggets/project-management-golden-triangle/

6.2 Threats to validity

For the analysis of the internal and external validity of our methodology and results, we employ the definitions provided by Zhou et al. [37].

6.2.1 Internal validity

The internal validity "seek(s) to establish a causal relationship, whereby certain conditions are believed to lead to other conditions" ([37]).

An aspect to consider is the quality of the search query employed. In order to include as many relevant publications as possible, we used several synonyms for the different aspects of our search and included as many databases as possible. However, due to the limited extent of the seminar and time constraints, we had to limit our research query to keywords in the abstract and title of the publications. A broader search query would have enabled a more extensive literature review.

It is also due to limited time that we could not perform the quality assessment proposed by Kitchenham et al. [26].

To avoid bias regarding the categorization of the identified modernization methods by the 6R's, we took different measures to classify the publications equally: we reported our findings amongst each other, discussed the categorization of individual publications together and examined parts of the publications categorized by the second author respectively.

A further aspect regarding internal validity is the publication bias of the reviewed publications. We have observed that the majority of the reviewed case studies describe the advantages of the selected modernization method only, rather than also specifying the drawbacks and challenges faced in the process. We can therefore only make suggestions on the disadvantages and lack of information on these approaches, as done in the discussion of results (section 6.1).

6.2.2 External validity

The external validity "define(s) the domain to which a study's findings can be generalized" ([37]). To achieve a reproducible result and minimize bias in the selected publications, we followed the methodology proposed by Kitchenham [26] and described the entire process, particularly the selection criteria. Similarly to the procedure for the categorization of the publications, we continuously compared our approaches and decisions regarding the selection of relevant publications.

7. CONCLUSIONS AND FUTURE WORK

Modernizing a legacy system can be a successful task if the appropriate method is selected. In this paper we conducted a comprehensive scientific literature review on experience reports in the field of legacy system modernization. We provide an overview of methods applied in the research community, as well as major arguments and justification authors stated regarding different methods. Finally, we proposed categorization based on our previous results that can be used as guideline for decision.

Further work could evaluate the proposed categorization based on the five dimensions for applicability, usefulness and completeness. Furthermore, conducting the SLR with a more inclusive search query could improve the quality and significance of the results.

8. REFERENCES

[1] B. Althani and S. Khaddaj. Systematic Review of Legacy System Migration. In *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*, pages 154–157. IEEE, 2017.

[2] Amazon Web Services. 6 Strategies for Migrating Applications to the Cloud | Amazon Web Services, 2016.

[3] Andreas Fürnweger, Martin Auer, and Stefan Biffl. Software Evolution of Legacy Systems - A Case Study of Soft-migration. In Slimane Hammoudi, Leszek A. Maciaszek, Michele Missikoff, Olivier Camp, and José Cordeiro, editors, *ICEIS 2016 - Proceedings of the 18th International Conference on Enterprise Information Systems, Volume 1, Rome, Italy, April 25-28, 2016*, pages 413–424. SCITEPRESS, 2016.

[4] W. K. G. Assunção, T. E. Colanzi, L. Carvalho, J. A. Pereira, A. Garcia, M. J. de Lima, and C. Lucena. A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices: An Industrial Case Study. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 377–387, 2021.

[5] L. Aversano, G. Canfora, A. Cimitile, and A. de Lucia. Migrating legacy systems to the Web: an experience report. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 148–157, 2001.

[6] P. Bhallamudi and S. Tilley. SOA migration case studies and lessons learned. In *2011 IEEE International Systems Conference*, pages 123–128, 2011.

[7] H. P. Breivold, S. Larsson, and R. Land. Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 232–239, 2008.

[8] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[9] W. C. Chu, Chih-Wei Lu, J. P. Shiu, and Xudong He. Pattern based software re-engineering: a case study. In *Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99) (Cat. No.PR00509)*, pages 300–308, 1999.

[10] S. Comella-Dorda, K. Wallnau, R. Seacord, and J. Robert. A survey of legacy system modernization approaches. page 30, 04 2000.

[11] S. Datar and S. R. Schach. Reuse of legacy software in object-oriented re-engineering: A case study. *Transactions of the South African Institute of Electrical Engineers*, 87(3):101–107, 1996.

[12] A. Dedeke. Improving Legacy-System Sustainability: A Systematic Approach. *IT Professional*, 14(1):38–43, 2012.

[13] Dipl.-Ing. Udo Thiele. Mainframe-Modernisierung – Best Practice im Überblick: Metamorphose der Dinosaurier. *ITSpektrum*, (04/2022):10–15, 2022.

[14] B. Düchting and T. Laszewski. Delta lloyd deutschland data migration case study. *Information*

Systems Transformation: Architecture-Driven Modernization Case Studies, 2010.

[15] J. Ewer, B. Knight, and D. Cowell. Case study: an incremental approach to re-engineering a legacy FORTRAN Computational Fluid Dynamics code in C++. *Advances in Engineering Software*, 22(3):153–168, 1995.

[16] M. Galinium and N. Shahbaz. Success factors model: Case studies in the migration of legacy systems to Service Oriented Architecture. In *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*, pages 236–241, 2012.

[17] B. Geppert and F. Rossler. Effects of refactoring legacy protocol implementations: a case study. In *10th International Symposium on Software Metrics, 2004. Proceedings*, pages 14–25, 2004.

[18] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution*, 14(1):1–30, 2002.

[19] W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff. The Dublo architecture pattern for smooth migration of business information systems: an experience report. In *Proceedings. 26th International Conference on Software Engineering*, pages 117–126, 2004.

[20] H. E. Hayretci and F. B. Aydemir. A Multi Case Study on Legacy System Migration in the Banking Industry. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12751 LNCS:536–550, 2021.

[21] J. Henrard, D. Roland, A. Cleve, and J.-L. Hainaut. An Industrial Experience Report on Legacy Data-Intensive System Migration. In *2007 IEEE International Conference on Software Maintenance*, pages 473–476, 2007.

[22] P. Jamshidi, A. Ahmad, and C. Pahl. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing*, 1(2):142–157, 2013.

[23] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented re-engineering of legacy systems into product line assets - A case study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3714 LNCS:45–56, 2005.

[24] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do professionals perceive legacy systems and software modernization? In P. Jalote, L. Briand, and A. van der Hoek, editors, *Proceedings of the 36th International Conference on Software Engineering*, pages 36–47, New York, NY, USA, 2014. ACM.

[25] R. Khadka, A. Saeidi, A. Idu, J. Hage, and S. Jansen. Legacy to SOA Evolution. In A. D. Ionita, M. Litoiu, and G. Lewis, editors, *Migrating Legacy Applications*, pages 40–70. IGI Global, 2013.

[26] B. Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[27] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: A case study. *Journal of Software Maintenance and Evolution*, 18(2):109–132, 2006.

[28] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. van Gorp. Refactoring: Current Research and Future Trends. *Electronic Notes in Theoretical Computer Science*, 82(3):483–499, 2003.

[29] P. Antonini, Gerardo Canfora, and Aniello Cimitile. Reengineering Legacy Systems to Meet Quality Requirements: An Experience Report. In Hausi A. Müller and Mari Georges, editors, *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994*, pages 146–153. IEEE Computer Society, 1994.

[30] A. Padenga. *Application Software Re-engineering.* Dorling Kindersley, 2010.

[31] Prabhakar Cherukupalli and Y. Raghu Reddy. Reengineering Enterprise Wide Legacy BFSI Systems: Industrial case study. In Srinivas Padmanabhuni, Raghu Nambiar, Premkumar T. Devanbu, Murali Krishna Ramanathan, and Ashish Sureka, editors, *Proceedings of the 8th India Software Engineering Conference, ISEC 2015, Bangalore, India, February 18-20, 2015*, pages 40–49. ACM, 2015.

[32] H. M. Sneed and K. Erdoes. Migrating AS400-COBOL to Java: A Report from the Field. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 231–240, 2013.

[33] Teta Stamati, Konstantina Stamati, and Drakoulis Martakos. Key Factors in Legacy Systems Migration - A Real Life Case. In Yannis Manolopoulos, Joaquim Filipe, Panos Constantopoulos, and José Cordeiro, editors, *ICEIS 2006 - Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration, Paphos, Cyprus, May 23-27, 2006*, pages 340–344, 2006.

[34] P. Tonella and R. Tiella. Weekly Round Trips from Norms to Requirements and Tests: An Industrial Experience Report. In *2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing*, pages 20–26, 2015.

[35] A. Vera-Baquero, O. Phelan, P. Slowinski, and J. Hannon. Open Source Software as the Main Driver for Evolving Software Systems Toward a Distributed and Performant E-Commerce Platform: A Zalando Fashion Store Case Study. *IT Professional*, 23(1):34–41, 2021.

[36] H. Wu, X. Qian, A. Shulman, K. Karanawat, T. Singh, H. P. Crowell, P. Bhimani, C. Tang, Y. Li, L. Zhang, and C. Ulherr. Move Real-Time Data Analytics to the Cloud: A Case Study on Heron to Dataflow Migration. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2064–2067, 2021.

[37] X. Zhou, Y. Jin, H. Zhang, S. Li, and X. Huang. A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 153–160. IEEE, 2016.