Received: 2 May 2019 | Accepted: 25 June 2019
DOI: 10.1002/smr.2217

SPECIAL ISSUE PAPER

WILEY Journal of Software: Evolution and Process

Opportunities for software reuse in an uncertain world: From past to emerging trends

Rafael Capilla[1] | Barbara Gallina[2] | Carlos Cetina[3] | John Favaro[4]

[1] Department of Informatics, Universidad Rey Juan Carlos, Mostoles, Spain
[2] School of Innovation, Design and Technology, Mälardalen University, Västerås, Sweden
[3] Computer Science Department, Universidad San Jorge, Zaragoza, Spain
[4] Innovation & Technological Services, Intecs Solutions, Pisa, Italy

Correspondence
Rafael Capilla, Department of Computer Science, Universidad Rey Juan Carlos, Mostoles 28933, Madrid, Spain.
Email: rafael.capilla@urjc.es

Abstract
Much has been investigated about software reuse since the software crisis. The development of software reuse methods, implementation techniques, and cost models has resulted in a significant amount of research over years. Nevertheless, the increasing adoption of reuse techniques, many of them subsumed under higher level software engineering processes, and advanced programming techniques that ease the way to reuse software assets, have hidden somehow in the recent years new research trends on the practice of reuse and caused the disappearance of several reuse conferences. Also, new forms of reuse like open data and feature models have brought new opportunities for reuse beyond the traditional software components. From past to present, we summarize in this research the recent history of software reuse, and we report new research areas and forms of reuse according to current needs in industry and application domains, as well as promising research trends for the upcoming years.

KEYWORDS
domain analysis, reusability, software asset, software reuse

1 | INTRODUCTION

Since the software crisis late in the 1960s and McIlroy's idea to envision the reuse of code and software components, it has been a long story where researchers and practitioners have experimented with different forms of reuse.

1.1 | The birth of software reuse

Software engineering celebrated its 50th anniversary in 2018. It was in 1968 that the NATO Conference was held in Garmisch, Germany where the term was born.[1] The anniversary was commemorated with a special issue in IEEE Software Magazine.[2] Consequently, 2018 was also the 50th anniversary of software reuse, because it was during that same conference that Doug McIlroy made the presentation, called "Mass Produced Software Components,"[3] that is considered to be the genesis of software reuse. The leader of the American delegation to that conference was Alan Perlis, another pioneer of software reuse who in 1987 coedited with Ted Biggerstaff the two-volume Software Reusability, which effectively consecrated reuse as a formal discipline.[4] Shortly after the publication of Software Reusability, a reference model for reuse known as the "3C model" was elaborated by a working group.[5] It obtained its first broad exposure in a publication in 1991, where it was stated that "...the 3C reference model has the potential to become the accepted basis for discourse on reusable software components among members of the reuse community."

The 3C model described in Figure 1 defines and distinguishes three ideas: concept, content, and context. [5,6]

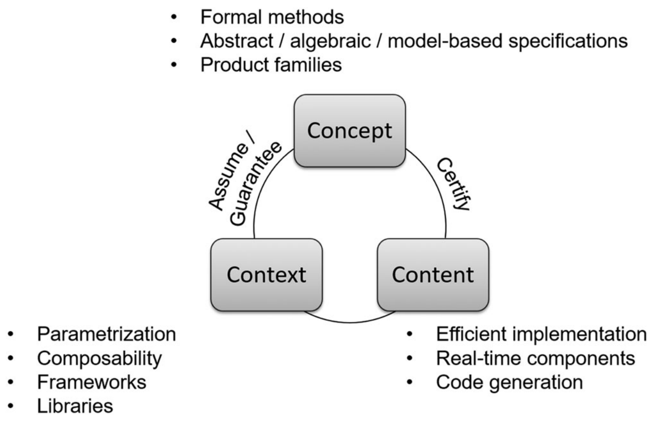

FIGURE 1 The 3C model contextualizing software reuse

- In the simplest terms, concept is a reusable component's specification, "what it is." This has led to formal component specifications and much more.
- Early on the key idea emerged of separating the specification from the implementation, the Content "how it works." The iconic example is a stack implemented variously as an array, a linked list, and so forth.
- Finally, the context indicates "how it is used." The explosion of the Ariane 5 rocket back in the 1990s was a famous case of a software component from the Ariane 4 being reused in the wrong Context.[7]

It is notable how this simple reference model became a kind of road map for the next 25 years of research in software reuse. Over those years, researchers made remarkable progress in all three dimensions of the reference model, producing libraries, frameworks, methodologies, generators, product families-an entire set of sub-disciplines that characterize software reuse as we know it today.

1.2 | The death of software reuse

Another pioneer in software reuse was Rubén Prieto-Diaz, who applied the ideas of faceted classification libraries to reuse repositories [Ruben1991], and also had significant involvement in the launching the field of domain analysis (DA).[8] Approximately 25 years after the seminal paper of McIlroy, at the Third International Conference on Software Reuse in 1994, Prieto-Diaz presented a paper entitled *The Disappearance of Software Reuse*,[9] which began with the statement, "I am looking forward to the day when software reuse is dead." (The accompanying presentation included a depiction of a tombstone inscribed with "RIP Software Reuse"). The central argument of the paper was that "...the ultimate success of software reuse will be marked by its disappearance. This disappearance will not come by elimination but by integration." This argument was accompanied by seven predictions:

1. Reuse will become systematic, formal, and integrated with software engineering;
2. Many organizations will establish their corporate reuse programs;
3. Software development environments and CASE tools will support reuse, DA, and domain engineering (DE);
4. A reusable parts industry for specialized domains will emerge;
5. Software development standards will evolve to support reuse within and across domains;
6. DE will be the dominant discipline in software engineering;
7. Software reuse will cease to exist.

It is fair to say (prediction 4) that much large commercial software development now involves large frameworks, as well as component libraries and connectors that permit the assembly of sophisticated applications in a short amount of time. The major software standards (prediction 5) now include requirements for reuse (cf. Section 3.4)—albeit cross-domain reuse somewhat less. The success of product line engineering (prediction 6) is there for all to see (cf. Section 3.1). This leaves us with the last prediction. The intended sense of Prieto-Diaz by that prediction has already been explained above, but in fact, this prediction did imply a death: the death of software reuse *research*. As the other predictions were gradually realized over the years, the question of whether reuse research would die no longer seemed unreasonable. And sure enough, over the last few years, there have been more and more people openly questioning whether there was any significant research in software reuse left to do, which is directly related to the decline in number of reuse conferences. It is the thesis of this paper that, to paraphrase Mark Twain, "reports of the death of software reuse research are greatly exaggerated." On the contrary, there is a whole new set of opportunities that are slowly but surely emerging for software reuse research. The new opportunities are coming from the growing ambition to practice software reuse in areas in which uncertainty is a significant defining characteristic.

1.3 | An uncertain world

For the first 25 years of the 3C reference model, there was always the implicit assumption that it was possible to precisely and completely specify a reusable component; that it was possible to completely understand the implementation of a component; and that it was possible to identify the complete context in which the component would be reused. This assumption of certainty in all three dimensions was also considered to be a necessary condition for doing reuse. But this assumption has been breaking down, not because of necessity, but because of a growing ambition to reuse in circumstances where this assumption does not hold where we don't necessarily know for sure what it is; or how it works; or how we will use it. This is true in all three dimensions of reuse.

The remainder of this paper is as follows. Section 2 describes main trends on software reuse research and practice over the past 30 years while in Section 3, we report on recent trends related to other software engineering practice areas. In Section 4, we discuss some promising application domains where software reuse is succeeding, and in Section 5, we report on the results of a survey with reuse practitioners to highlight the role of software reuse in industry. Finally, Section 6 outlines a research road map for the future of reuse, and we draw conclusions in Section 7.

2 | EARLY FORMS OF SOFTWARE REUSE

For more than 30 years, software reuse was a popular software engineering research field and practice area in industry. From the visionary ideas of McIlroy up through the 1990s, several forms of reuse were investigated. In this section, we present the major forms of software reuse that were popular in conferences like the Workshop on Institutionalizing Software Reuse (WISR), Symposium on Software Reusability (SSR), the European Reuse Workshop (ERW), and the International Conference on Software Reuse (ICSR), as major software reuse research events. Instead of a long report about past research on software reuse,[10] we summarize under the main achievements under the following four major categories.

2.1 | Domain analysis

The role of DA techniques has been crucial for understanding reuse of high-level abstraction artifacts and knowledge. DA was initially understood as a technique to *identify the set of relevant objects and operations in a particular domain.** Hence, building and reusing software in "unknown" or complex domains (eg, critical systems), DA was proven as a human and complex technique to understand the terminology, elements, and processes for building and reusing software systems and, in particular, a way to understand a particular domain by software engineers who must rely on the domain experts. From early DA methods [Arango 1991] like Prieto-Díaz's Sandwich method[8] where the identification of the relevant objects and operations were critical for knowledge reuse, the classification of the terms discovered (eg, faceted classification[8,11]) and the implementation of the processes and operations for engineering complex systems, many of them with incipient object-oriented techniques, was crucial for modern software reuse practice in industry. Over years, several DA processes have been proposed, but most of the authors failed to explain how to carry out such human and complex manual activities, such as knowledge reuse.[12,13] In order to bring DA activities closer to other software engineering processes, the so-called DE process[14,15] encompasses DA with the development and reuse of different software artifacts in a more systematic way for producing reusable information. Approaches like Domain Analysis and Reuse Environment (DARE)[16] were proposed years later to integrate DA processes under CASE tools and exploit the commonalities and variabilities of software systems and capturing the domain information from experts, documents, and code. Hence, such commonalities and variabilities are the basis of current feature modeling approaches.

Nevertheless, the importance of capturing the essence of software systems and modeling the problem domain is one of the main reasons why these techniques still survive nowadays. Although DA practice enables the identification of the relevant domain elements and system features that can be reused across different applications, the majority of reuse approaches concentrate on the code and architecture rather than domain models.

2.2 | Generators and transformation models

One complementary way to achieve software reuse in addition to component and code reuse is to employ generative and transformation techniques. For many years, code generators have been widely used to produce executable code based on reusable code that is customized via

*A domain represents an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

templates. Early approaches like DRACO,[4,17] a methodology for engineering reusable systems and GenVoca[18] (a generator for defining high level constructs in a high level language) for synthesizing software systems pioneered transformational approaches. A second generation of these tools produced popular 3GL and 4GL generators (eg, CLIPPER and DATAFLEX) aimed at transforming abstract into concrete programs leveraging code reuse and speeding up the task of programmers. In addition, significant effort has been expended to create a variety of domain-specific languages and code generators using more modern techniques like Model-driven Engineering (MDE) approaches, which are still used to increase reuse when transforming code constructs from one programming language to another. However, as writing code generators and transformation rules is difficult and costly, many of the today's code generators have been replaced by reusable libraries and components. Other generative methods and tools combined with component-based development as approaches to leverage code reusability can be found in Czarnecki and Eisenecher.[19]

2.3 | Components, languages, and code reuse

Several high level artifacts like reusable architectures, software schemas, and abstractions in high and very high level languages (VHLLs) such as Smalltalk, ML, SETL, or MODEL have been attempted over the years, such as described in Krueger.[10] However, code and component reuse has proven to be easier and more productive than other higher level reusable artifacts.

Code reuse is one of the most primitive forms of reuse as a way to avoid writing duplicate code and committing programming errors. Reusing code is not a complex practice if done systematically (as opposed to “ad-hoc” reuse). However, producing reusable code is costly and difficult, as the quality of the code must be extremely high, with zero programming faults and time-consuming testing that the reusable code does not fail. In addition, interoperability issues when connecting the reusable code with other code pieces in the system are a major concern. According to Biggerstaff and Perlis,[4] several code reuse forms were proposed years ago. From subroutine libraries, compilers, and parameterized systems, the spectrum of reusability has been widening to envision McIlroy's idea. Along with the evolution of programming languages, software reuse has evolved since the appearance of C++ to provide additional facilities for programmers for reusing code. Therefore, encapsulating larger functionality in code components has changed the paradigm of software technology for *programming in the large* more complex systems. Reuse technology has since evolved from reusing small subroutines to large object-oriented (OO) components encompassing significant functionality that can be customized for different application types (eg, reusable APIs to create customized interfaces). To make reusability attractive, it becomes necessary to reuse with information hiding where the complexity of large OO components is hidden from the programmer via interfaces and parameterized modules to adapt the reusable code to different needs. The story of reusable programming languages is a long one, from older languages like CLOOS, OBJ, ADA, and Smalltalk to more recent ones like C++, Java, or Python. A retrospective look at several reuse experiences and case studies can be found at Biggerstaff and Perlis.[20]

One popular form of reuse used for years was the Commercial Off-The-Shelf (COTS) approach, involving the selection and use of COTS components as *products*, which could easily be installed and interoperate with existing software components at a relatively lower cost. Variations of COTS were also created, such as modifiable off-the-shelf (MOTS) components (customizable assets to be adapted to any context). COTS developed for specific organizations (eg, government off-the-shelf and NATO off-the-shelf) were also developed and included in product-line approaches and service-oriented architectures.[21] Nevertheless, problems associated with integrating COTS from different vendors, as well as problems with controlling the evolution of COTS based systems due to a lack of support of COTS components, have been major obstacles to achieving greater market penetration of COTS technology.

With the appearance of object-oriented programming (OOP), reuse of code artifacts evolved into component reuse and component-based software engineering (CBSE) techniques where the reusable code was encapsulated in bigger reusable modules, libraries, and frameworks. From early OO languages like SIMULA and Smalltalk in the 1960s to C++/C# and VisualBasic in the 1980s and more recently Java, the Microsoft .NET platform or Python, reusability via software components was achieved through increased modularity, interoperability via standard interfaces, and robustness. Therefore, the visionary idea of McIlroy[3] about software produced massively using reusable software components was realized to the fullest.

2.4 | Reuse cost models

Given that one of the primary arguments put forward for software reuse has always been economic, it is not surprising that research in this subdiscipline began early.[22] One of the earliest and most influential attempts at constructing a reuse cost model originated at the Software Productivity Consortium, which based the cost model on the economic principle of *time to payback*—that is, the amount of time needed to achieve a break-even situation.[23] A payoff threshold value was defined based upon estimations of the relative cost of developing a component for reuse and the relative cost of integrating the component. This cost model was then used to analyze reusable components which varied in complexity and integration cost, yielding correspondingly varied results in the time to payoff.[24] Subsequently, cost models of increasing sophistication were introduced, in particular by Poulin[25] who explored the use of standard financial metrics such as the internal rate of return in order to bring software reuse into a more standard business context. A prominent exponent of this business-oriented approach was Lim, who introduced the widely accepted financial measure of net present value and gave examples of its implementation in a reuse context.[26] The alignment of the study of software reuse economics with standard business principles brought about an interest in strategic aspects of software reuse—that is, the support that reuse could give to a company's market strategies. This led in turn to an application of Value Based Management to standard reuse management theory, and culminated in Value Based Software Reuse Investment, in which the use of real options was investigated for the analysis of the benefits of reuse for supporting strategic flexibility.[27] Interestingly, this line of investigation ultimately led a few years later to a broadening of the application of value-based principles to the whole of software engineering. [28]

2.5 | Summary

Unfortunately, the integration of most of the reuse forms under more modern software engineering practices like component models and software product line (SPL) engineering practices brought about the disappearance of the majority of software reuse conferences except for ICSR, which has been renewing topics and papers to adopt more modern forms of reuse, as we discuss in next section.

3 | CURRENT AND NEW TRENDS IN SOFTWARE REUSE

Today, many software reuse processes have been subsumed or integrated under more modern software development approaches, and today's application domains for building software have changed. For example, contextual knowledge is widely exploited by autonomous and context-aware systems (eg, robots and IoT); open source software systems have emerged as an evolution of COTS systems; and data are driving industry 4.0 solutions (eg, Big Data and Open Data solutions). In this section, we describe four current major trends where reuse is a common goal for modern development approaches and systems.

3.1 | Product lines, features, and context analysis

Clements and Northop[29] define an SPL as "a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." This notion has been reused in other contexts. For instance, in process engineering, the notion of *process line*[30] and its extension to safety-critical systems engineering[31] was introduced.

A recent survey[32] reveals that most SPLs are built when there are already products. These legacy products are usually, for instance, similar product variants that were implemented using ad-hoc reuse techniques such as *clone-and-own* (CaO).[33] CaO refers to the practice of reusing software by copying and adapting parts from legacy software products to create a new software product. Version control systems, such as SVN and Git, have contributed to the popularization of CaO by means of their branch and merge functionality.

More and more approaches perform feature location to extract reusable assets. This is the case of approaches for extractive SPL adoption (ESPLA).[34] These approaches initiate the SPL through capitalization on existing software products. In other words, products created under CaO settings are reengineered into a family of products where reusable assets are formalized.

The public catalog for ESPLA case studies[35] reveals that the potential of reuse is not only restricted to source code: Documentation, designs, models, and components are examples of software artifacts that are also being reused. A recent comparison of ESPLA approaches[36] identifies the following approaches as the ones with support for reusable asset extraction: Pure::variants var. extractor,[37] BUT4Reuse,[38] and FLiMEA.[39,40]

The most general definition of the notion of *feature* was given by Kang et al:[41] "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems." Czarnecki and Eisenecker focus on the SPLs variants: "a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family." Finally, Bosch [42] places an emphasis on system behavior: "a logical unit of behavior that is specified by a set of functional and quality requirements." Despite these system-centered definitions, the notion of feature has been stretched in various ways in order to denote characteristics not necessarily belonging to system software only but to any other artifact at any abstraction level of system engineering, including the process model describing the system engineering itself.

It is widely recognized that DA is one of the important activities in current SPL practice during the DE phase. However, the need to model system features and the fact that more and more systems applications are context-dependent (eg, smart homes, wearable devices, and autonomous cars) has brought about the need to model "context properties" as part of the DA process. Hence, some approaches suggest renaming old DA processes as "context-oriented DA" or just simply "context analysis" where context properties are modeled alongside with other context features and reused during the application engineering phase. For instance, in Desmet et al,[43] the authors highlight the role of modeling context-aware systems and their behavioral variations using a Context-Oriented Domain Analysis (CODA) approach to understand and model context requirements in order to support all possible combinations of context adaptations. Recent approaches describe the role and reuse context features[44] for context-aware applications and how the introduction of these context features (ie, depicted in the same or in separate feature models) in modern context-aware systems like the automotive domain[45] can increase the reusability of systems features and manage adaptive behavior at runtime.

3.2 | Open data reuse

The idea behind open data is to have data freely available to everyone to use. Therefore, under the main umbrella of open-source software (OSS), open data, or *linked open data* (LOD) as a more structured way to publish and query data is becoming an opportunity for government organizations to publicly share and reuse data. Although the idea of open data is not new, it is appearing now as a new form of software reuse in different application domains and applications (eg, government, medical, scientific, and transportation) and increasing collaborative facilities that add value to the data. Initiatives like the EU Open Data[†] Portal as a main entry point of access to public data including the European standard DCAT application profile for data portals (DCAT-AP) [‡]promote open data by default, accessible and usable, and achieving semantic interoperability for public services. As reported by the European Data Portal, the potential value of open data reuse will increase in the coming years as more and more open data providers become active. The open data chain will add value for open data reusers and aggregators of knowledge. Without going into the technical goals of data formats and applications for achieving semantic interoperability of data sets (ie, using different data formats like CSV, JSON, or XML), there is a new opportunity to reuse different kinds of artifacts that increase the value of published data and create new research paths for organizations—for example, developing open data reusability metrics (eg, MELODA[§]) to assess the degree of homogeneity in published datasets in six different dimensions.

3.3 | Services and micro-services

Reusable web services have became very popular since the 1990s for building part of the functionality of the system (ie, often the noncritical parts) in the form of reusable components invoked using standard Internet mechanisms like SOAP, REST, and WEB-RTC. The use of common data formats like XML (eg, WSDL) or JSON to represent the data exchanged by these web components, together with languages like WS-BPEL to compose services, has facilitated and modernized the reusability of many legacy applications like banking systems. Software companies like Amazon and Google have popularized the use of these services so developers can invoke Amazon's web (ie, AWS [¶]) services and Google APIs (eg, Google Maps API Web Services [#]) for creating reusable serverless applications. Therefore, older approaches like JavaBeans (reusable classes encapsulating many objects into one single object) and its extensions for building enterprise software (ie, Enterprise Java Beans [EJB]) using APIs for modular construction of more complex systems have been replaced in many cases by web service technologies that promote loosely coupled applications, higher reusability, and ease of use, and whose lightweight common data formats have facilitated the interoperability of distributed applications.

Today, the trend toward using services continues. Recently, it has evolved toward *microservices*, whereby for certain systems the size of the reusable web services has been reduced to just a few lines, leading to a new architectural style called *microservice architectures* (MSA). As a consequence, monolithic architectures and systems are now engineered (or migrated) as an MSA or pattern [‖], in favor of better deployability, testability, extensibility, scalability, and reusability (among other relevant quality properties) and (re)-using small pieces of code. Well-known industry cases like Netflix or eBay are examples of successful MSA that decompose systems according to business capabilities.

Modern software platforms like Docker [**]enable the deployment of microservices for building large-scale applications and use, for instance, Docker containers (ie, a standardized unit of software) to build a microservices architecture. Compared with other traditional reusable components, the size of microservices does not actually matter. Rather, what is perceived to be more important when adopting a microservices architecture is modularity, ease of development and test, scalability, and resilience of the solution to architecture erosion.

3.4 | Asset reuse in safety-critical systems engineering: In search of a pedigree

An *asset*[46] is defined as an item that has been designed for use in multiple contexts. An asset, in the context of this paper, refers to existing software. Designing software for multiple contexts is however not an easy task in the context of safety-critical systems engineering. The failure that in 1996 caused the explosion of Ariane 5 [1] is an example of a faulty design. As known, Ariane 5 contained a piece of code that was appropriate for its predecessor Ariane 4 but that turned out to be catastrophic for Ariane 5. As observed by Gallina,[47] as a consequence of the Ariane 5 accident, asset reuse in safety-critical systems engineering has been considered a taboo . On the one hand, the "to-reinvent or not-to-reinvent" dilemma has been often solved by a tacit acceptance that starting from scratch is safer. On the other hand, standardization bodies have worked hard in order to

[†]http://data.europa.eu/euodp/en/home

[‡]https://joinup.ec.europa.eu/solution/dcat-application-profile-data-portals-europe

[§]MEtric for assessing open DAta sources: https://www.meloda.org/about-meloda/

[¶]https://aws.amazon.com

[#]https://developers.google.com/maps/documentation/directions/web-service-best-practices

[‖]https://microservices.io/patterns/microservices.html

[**]https://www.docker.com/solutions/microservices

develop the necessary requirements for guiding (a) the reuse of pre-existing "pedigree-less" software, (b) the development of "pedigree-equipped" (safely reusable) software, and (c) the development of plug and play platforms.

At the cross-domain level, the meta-standard IEC 6[48] introduced the notion of Software of Unknown Pedigree or Provenance (SOUP). Concerning the assessment of SOUP, IEC 61508, part 7, states: "In order to assess the safety integrity of the new system incorporating the pre-existing software, a body of verification evidence is needed to determine the behavior of the preexisting element. This may be derived (1) from the element supplier's own documentation and records of the development process of the element, or (2) it may be created or supplemented by additional qualification activities undertaken by the developer of the new safety related system, or by third parties. This is the *Safety Manual for compliant items* that defines the capabilities and limitations of the potentially reusable software element." IEC 61508 incorporates results developed by Bishop et al.[49]

In the medical domain, IEC 6[50] the standard for life cycle processes for medical device software, refined the notion of SOUP in the context of medical devices to denote pre-existing software that has not been developed to be integrated into medical devices, or software with unavailable or inadequate records of its development processes. IEC 62304 has also introduced guidance to treat SOUP.

In the aerospace domain, Integrated Modular Avionics (IMA)[51] consists of a flexible hardware and software platform with core functionality that can be easily extended and customized to meet specific requirements. An IMA platform is composed of modules, which are designed to be reusable in order to reduce development costs and occasionally facilitate certification programs. Some modules provide only mechanical functions (eg, cooling); others include core software and associated computing capabilities. IMA enables the use of legacy systems because components are interoperable and "plug and play." In addition to IMA, the Future Airborne Capability Environment (FACE)[52] represents a standard for software computing environments and interfaces designed to support the development of portable components across the general-purpose, safety, and security profiles. The standard enables developers to create and deploy a wide catalog of applications for use across the entire spectrum of military aviation systems through a common operating environment.

ISO 26262:[53] the standard for functional safety within the automotive domain, mentions reuse explicitly in some of its parts: Part 3 (concept phase), together with part 8 (change management), indicates what should be done in case of a modification of a pre-existing system in order to determine the impact on the system-related work product as well as in terms of process-related documentation (how to tailor the life cycle). Part 8 and part 10 introduce novel notions. Specifically, part 8 introduces the notion of *proven in use* argument for reusing pre-existing elements, in line with the previous standards, where guidance for the reuse of pre-existing software is provided. Part 10 (clause 9) introduces the notion of the *Safety Element Out of Context* (SEooC), defined as a safety-related element that is not developed in the context of a specific item (ie, it is not developed in the context of a specific road vehicle). Part 10 also recommends corresponding processes for software and hardware element development and integration. These processes provide, on the one hand, guidance to suppliers on how to develop safely reusable elements, and on the other hand, guidance to integrators on how to reuse components developed out of context. The notion of SEooC and the corresponding development process have the potential to play a key role in pedigree-equipped engineering. SEooC may be thought of as a re-interpretation of the 3C model in the context of safety-critical systems engineering. In the automotive domain, not only the reuse of individual components is of primary interest but also the reuse of the software architecture. AUTOSAR,[54] which stands for AUTomotive Open System ARchitecture, is a standard that pursues the objective of creating and establishing an open and standardized software architecture for automotive electronic control units (ECUs), aimed at, among other objectives, achieving a higher penetration of COTS components across product lines. AUTOSAR is also in search of a pedigree, and several efforts have been made to integrate ISO 26262, part 6, requirements in order to make AUTOSAR ISO 26262-compliant (see previous study[55] for the extension of AUTOSAR with safety). AUTOSAR represents the automotive counterpart of IMA in avionics.

Finally, it is relevant to mention the space domain, which in these decades has learnt its lessons and has aligned itself with some of the previously mentioned domains. In the space handbook of software reuse,[56] existing software embraces different software artifacts since reuse can take place at different phases of the software life-cycle. The artifacts are requirements, components, modules, libraries, and source code, as well as documents, plans, tests, and data. To be able to reuse existing software, also in the space domain a "pedigree," called *product service history*, is considered to be of value and must be carefully analyzed in conjunction with the configuration changes and the process examination, whose objective is to establish the validity and value of the records of the existing software.

4 | APPLICATION DOMAINS

In this section, we describe some sample application domains exhibiting cases of successful reuse. Although the list is not complete, it provides an overview of where different forms of reuse are more promising.

4.1 | Reuse in the automotive domain

Reuse in the automotive domain has been largely explored and used, separating the hardware of the ECUs from the software embedded in it.[57] Standard core assets and network drivers for the Controller Area Network (CAN) bus have been developed as reusable assets independently of the hardware but able to exchange data with the ECUs via a data bus and dedicated interfaces. However, software of modern cars is becoming more and more complex every day, and the traditional AUTOSAR reference architecture[54] must confront new challenges like building software for autonomous vehicles, night vision systems, pedestrian detection systems, and so forth. Therefore, the challenge for software builders to connect autonomous and smart vehicles to other cars and road sensors demands not only 5G communications but also open and flexible architectures to cope with new development challenges and continuous software updates to support the vehicle infrastructure. Companies like Systemite AB (Sweden) facilitate configuration, traceability, and version control of models by enabling reuse of specific configurations in as many contexts as needed on behalf of tools like SystemWeaver.[††] Therefore, typical reusable assets for this domain are software components, models and SIL (software-in-the-loop) code used by MATLAB/Simulink tools for simulation and verification tasks and building control algorithms (eg,[58]). Atypical but of increasing relevance is also the reuse of arguments to speed up compliance with ISO 26262, while arguing about safety in the context of safety-critical product lines (eg,[59]).

4.2 | Reuse in the space domain

Despite the calamitous failure of the Ariane 5 rocket described earlier, reuse has long been vigorously pursued in space programs, and its standards have been updated to handle reuse as discussed in Section 3.4. Space systems have a key architectural characteristic that is essential to understanding the types of reuse that are pursued in those systems: Most space systems have a space segment—the system that is actually sent into space—-and a ground system—the system that controls the operations. The types of reuse pursued in each of these segments can be quite different. In the space segment, embedded systems with limited memory and CPU power are featured but with extremely high requirements on reliability and longevity. Reuse is considered an important weapon in the maintenance of the onboard (space) segment of a mission, where by a standardized onboard reference, architecture is used to allow the creation of software modules that can easily be replaced while the system is in space. An early example is Favaro et al.[60] A more recent example is Panunzio and Vardanega.[61] Ground systems, in contrast, contain enormous amounts of software that is, however, mostly less mission-critical in its nature. Here, there is very high interest in the reuse of open source software on standard platforms (eg, Linux), as well as large components for data display and processing. Finally, since space missions are often similar to one another, with variations such as destination and experimental purposes, many space Prime Contractors have initiatives to create space program families that can be managed like product lines, thereby reducing the amount of new work that must be invested in creating a new mission. Within the AMASS research project,[62] variations in space software systems as well as in processes, used to develop those systems, have been in focus and family-oriented solutions have been proposed to manage variants and impact analysis at process as well as system level.[63-65] The international CCSDS, a committee of global space agencies, has named reuse in all of its forms as one of its highest priorities. [66]

4.3 | Reuse in the rail domain

The European Railway Traffic Management System (ERTMS)[67] is an international standard, which contributes to the improvement of the interoperability, performance, reliability, and safety of modern railways. ERTMS relies on the European Train Control System (ETCS), an Automatic Train Protection system which continuously supervises the train, ensuring that the safety speed and distance are not exceeded. ERTMS/ETCS can be specified in one out of four main levels of operation, depending on the role of track-side equipment and on the way the information is transmitted to/from trains. ERTMS was conceived to enable interoperability among national networks, ie, to enable trains crossing national borders without requiring time-consuming and costly locomotive change. Given its different levels and its high degree of variability within each of them, ERTMS/ETCS constitutes an ecosystem. In the context of the Multiannual Programme of the Shift2Rail Joint Undertaking Initiative,[68] which is the innovation program in the rail sector under which various projects are funded, reuse is also an important implicit objective. Within some of the funded projects, for instance, product line engineering approaches are being applied,[69] and modular platforms are being conceived for future highly integrated and reconfigurable parameter-driven architectures, which require high levels of software abstraction with support for incremental certification, and rely on new advanced networking approaches. [70]

4.4 | Reuse in the home appliances domain

In the last decade, the home appliances domain has experienced a fundamental software-based transformation. More and more everyday home appliances are being run on software and augmented with online services. For instance, the newest induction hobs feature full cooking surfaces, where dynamic heating areas are automatically calculated and activated or deactivated depending on the shape, size, and position of the cookware placed on top. There has been an increase in the type of feedback provided to the user while cooking, such as the exact temperature of the cookware, the temperature of the food being cooked, or even real-time measurements of the actual consumption of the Induction Hob. All of these changes are made possible at the cost of increasing the software complexity.

[††]https://www.systemweaver.se

In the face of this ever-growing importance of software in the home appliances domain, industry is reengineering their product firmware into SPLs. An example of this is the induction division of BSH, the leading manufacturer of home appliances in Europe. BSH produces induction hobs under the brands of Bosch, Siemens, and Gaggenau (among others), and software is the heart and soul of its induction hobs. Reengineering firmware into SPLs was possible through feature location. These feature location efforts [39,40,71,72] range from Information Retrieval to Machine Learning, and include the dimension of Search-based Software Engineering.

Still, this domain faces challenges in the reengineering toward SPLs. Feature location is a challenge in these industrial environments where a vast amount of firmware is accumulated over the years, and this firmware has been created and maintained by different software engineers. Even though no single software engineer has a full understanding of the entire firmware, several software engineers can collaborate to complement each other's (partial) knowledge of the firmware. [73]

5 | WHAT DO PRACTITIONERS THINK?

In order to acquire practical insight from reuse practitioners and researchers, we carried out an online survey, which was sent to around 120 people involved in different forms of reuse activities. We took inspiration from a similar survey conducted in 1993, which posed 16 questions.[74] We renewed some the questions according to present circumstances and prepared a survey with 14 questions. Despite the low number (16) of responses, they provide qualitative insight into what practitioners are currently doing with respect to reuse activities. From the demographics, the 88.2% of the responses belong to Europe, the 5.9% to United States, and 5.9% from Asia. The ages of the respondents vary from 34 to 64 years (not all the participants answered this question), and most of the organizations are large companies or research centers (ie, number of employees from 10 000 up to 400 000+), along with a few small companies. The number of years of experience as professional software engineers varies from 5 to 34 years, so all can be considered senior, and the number of years of experience working with software reuse ranges between 3 and 30 years. Figure 2 shows the most popular application domains where reuse seems to be more successful. The "other domains" label belongs to those responses where the participants did not indicate a specific application domain but rather the type of reusable assets (eg, knowledge) or a process where reuse is used (eg, product line engineering or safety critical systems). As we can observe from Figure 2, the automotive and the aerospace domains described in Section 4 are very popular in the responses given by the respondent.

According to the type of reusable assets (**Q1**), the participants expressed their preferences for source code and components as the most popular ones; but also requirements, designs, and test cases are frequently reused, as Figure 3 shows. In this figure, a single response from one participant can include several assets, so we counted them as a different response. However, we decided to group in a different category subjects that answered "all assets," as we are unsure whether they include all the other types.

While the majority of the subjects considered software reuse valuable, a bit more than half of the subjects perceived that the programming language does not affect reuse (**Q2**). This is an interesting conclusion, as some programming languages seem to be more suitable than others for building reusable assets. The variety of programming languages used to engineer reusable assets is broad: The data indicated no language that seems more popular than others. Languages not only Haskell, C++, C#, Simulink, Java, and Python are commonly used but also others like C, Angular, or Go lang.

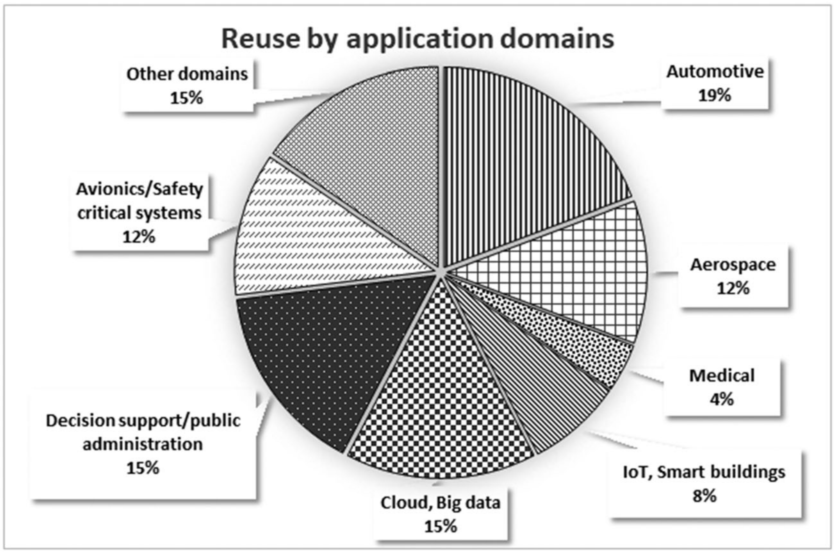

FIGURE 2 Most popular application domains of current reuse practice

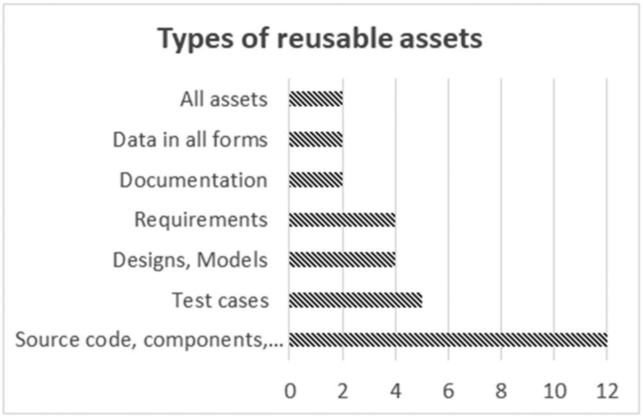

FIGURE 3 Types of reusable assets most commonly used

In addition, almost all the respondents prefer to reuse rather than build from scratch (**Q3**), and the supporting reasons are wide ranging. Some of the responses include "Easier to pick something than create something," "If good design (modular) reuse otherwise build from scratch," or "more productivity." Is it hard to categorize all the diversity of responses in favor of reuse, but the argumentation is along the lines of increasing speed and productivity and providing good designs as a way to understand better the code. Regarding reuse using open source practice (**Q4**), 47% of the respondents indicated they are not involved in reusing OSS components, so it is supposed that commercial solutions are preferred. However, all the respondents think the use of repositories increases reuse (**Q5**), such as Gitlab, SVN, or proprietary ones.

With respect to architectural concerns (**Q6**), the majority of the the respondents agreed that reuse influences the adoption of new architectural styles and expressed preferences for Web or Cloud based systems. However, 53% of the respondents did not provide any concrete style.

The economic point of view of software reuse practice is quite important for launching a systematic reuse process. Therefore, 82.4% of the participants think reuse is economically beneficial for their organizations (**Q7**), but few respondents were able to estimate the upfront investment cost (eg, ranging from 5% of the total effort to vast amounts) and the potential benefits (eg, "increasing speed," "at least double the cost," or "30% of the investment"). In addition, five of the respondents believed that recognition awards increase reuse in their organizations (**Q8**).

The respondents indicated also the type of the reuse strategy adopted in their organizations (**Q9**), so institutionalized practices are the most preferred ones to systematize reuse but, surprisingly, 47% of the subjects still adopted an ad-hoc reuse practice, such as shown in Figure 4.

The level of integration of reuse processes in other software engineering activities (**Q10**) is described in Figure 5. As we can see, 41% of the subjects said there is a full integration of reuse in their development processes while a subset of the answers reveals some kind of integration.

With regard to other nontechnical concerns, 70% of the subjects think that legal or license problems do not inhibit reuse (**Q11**), while 59% believe that standards facilitate reuse practice (**Q12**). The most popular application domains where the participants think reuse is more promising (**Q13**) map with those shown in Figure 2, but some others like *embedded systems*, *critical systems*, *cyber physical systems*, or *mechatronics* are highlighted as promising areas. Finally, question (**Q14**) refers to the quality properties the subjects think are more valuable when adopting a reuse strategy, and the answers are shown in Figure 6. According to the figure, maintainability and reliability are the most popular properties, but

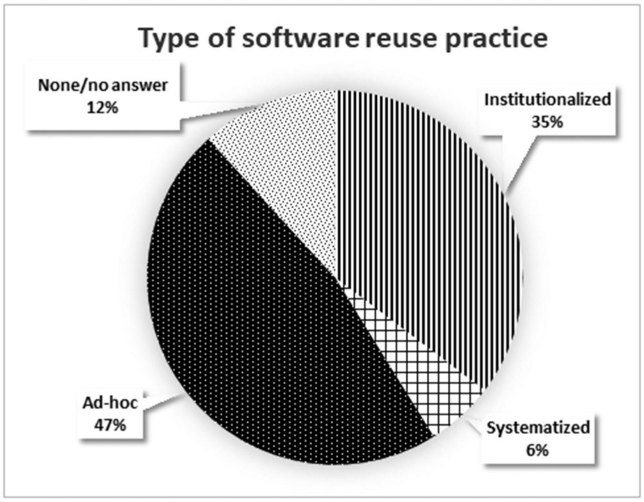

FIGURE 4 Type of reuse practices adopted by organizations

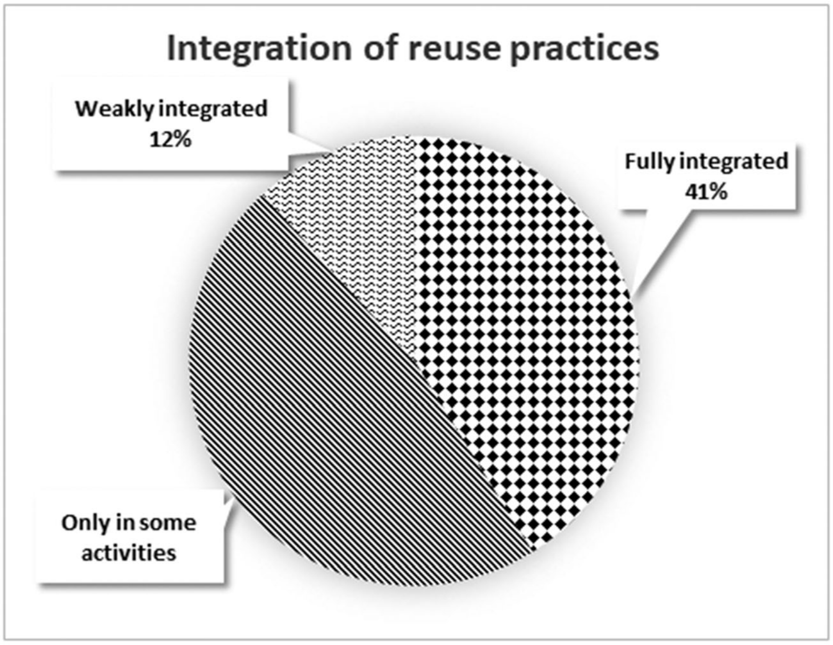

FIGURE 5 Integration level of reuse processes in software engineering activities

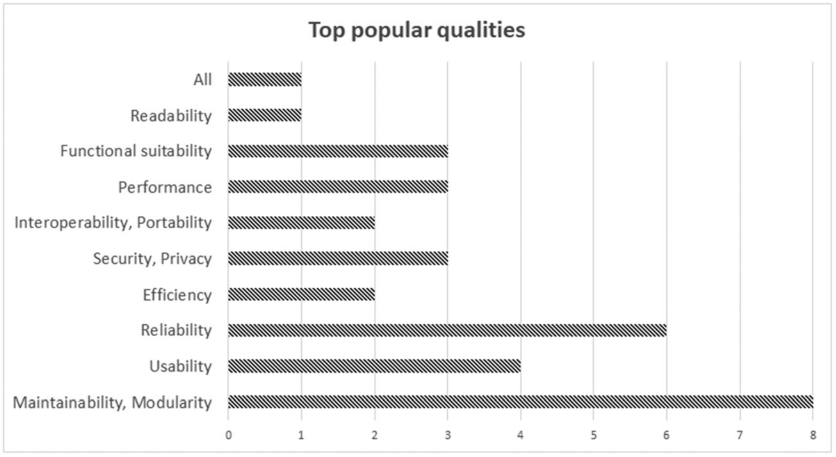

FIGURE 6 Most popular quality properties when adopting a reuse strategy

performance and functional suitability (ie, according to ISO 2500, the degree to which a product or system provides functions that imply needs when used under specified conditions) seem to be also very relevant.

With respect to Frakes and Fox,[74] we provide in Table 1 a comparison of the responses between the old survey and ours for nine questions that are nearly identical. As we can observe, there are interesting findings along these 26 years that we can summarize. First, the types of reusable artifacts have changed significantly, due to the evolution of programming languages. Second, half of the developers or software engineers still think that the programming language affects reuse. However, we observed a big shift from the first survey, where developers prefer to build their own software, compared with today, where the 76% of the respondents are in favor of reuse. In addition, while the respondents in the 1993 survey felt that rewards were rare for reuse, only 29% of today's respondents think the same way. Regarding the issue of having a common software process to promote reuse, the majority of the respondents disagreed. However, 41% of the respondents think today that software reuse must be fully integrated with other software engineering processes, and 47% expressed that only some integrated in some activities. In both surveys, most of the subjects think that legal problems inhibit reuse, and so this remains a major barrier for adoption. Surprisingly, another big shift concerns the use of repositories: from Table 1, we can observe the significant change since 1993 in favor of using repositories to increase reuse. Finally, as the variety of domains for reuse practice is great, it is difficult to predict new areas for successful reuse, but we can see that the aerospace domain is still very important and the increasing importance of software reuse in the automotive domain, mainly caused by the shift toward using ECUs to replace mechanical control.

TABLE 1 Comparison between two reuse surveys

Question Survey 1993	Response	Question Survey 2019	Response
How widely reused are common assets?	Unix tools, Document templates, X widgets, Ada math library	Which reusable assets are actually used?	Source code, components, libraries, test cases, designs, requirements
Does programming language affect reuse?	Divided	Does the programming language affect reuse?	Yes (47%)
Do developers prefer to build from scratch or to reuse?	Build their own software	Do developers prefer to build from scratch or to reuse?	Reuse (76%)
Does perceived economic feasibility influence reuse	High perception of economic viability correlated with source code reuse	Is reuse perceived to be economically beneficial for your organization?	Yes (82%)
Do recognition rewards increase reuse?	Rewards for reuse are rare	Do recognition rewards increase reuse in your organization?	Yes (29%)
Does a common software process promote reuse?	Disagree in most cases	To what extent is software reuse integrated into other software engineering processes in your organization?	Fully integrated (41%), only some activities (47%), weakly integrated (12%)
Do legal problems inhibit reuse?	Not an impediment for most respondents	Do legal or license problems inhibit reuse?	No (71%)
Does having a reuse repository improve code reuse	Having reuse repositories does not improve levels of code reuse	Do you think repositories increase reuse?	Yes (88%)
Is reuse more common in certain industries	Software (34%), aerospace (25%), manufacturing (14%)	In which application types / domains do you think reuse is more promising?	Automotive (19%), aerospace (12%), other domains (15

6 | A RESEARCH ROAD MAP FOR THE FUTURE

Based upon the results of of our survey, the most promising application domains for reuse and the different types of reusable assets of modern software systems, we summarize in this section some trends that exhibit considerable potential today and contribute toward the creation of a new road map for future reuse research. All of them represent, in different ways, a stretching of the three dimensions (the 3Cs) to deal with new uncertainties arising in modern applications.

6.1 | Trend 1. Knowledge reuse

The rapid advance of Big Data and Artificial Intelligence noted earlier has led to a trend toward the reuse of artifacts at very high semantic levels of abstraction, which might be termed "knowledge components." One concrete application is in the area of business innovation processes. Duggan[75] has defined an innovation process based upon the concept of so-called *historical precedents*. The process calls for a systematic search for historical precedents that represent potential solutions to partial aspects of an innovation problem, followed by their creative combination to arrive at a full solution. The possibility of creating a repository of historical precedents for search and retrieval has been investigated,[76] whereby as a first step, the utilization of semantic Wiki technology[77] is postulated. The process poses several major research challenges: the semantic fuzziness of the representation of the precedents, the fact that the insight contained within a precedent may change according to the specific innovation problem, and the difficulty of formulating search and retrieval queries. More powerful search and retrieval techniques such as latent semantic indexing are being investigated in this respect. This is an example of dealing with uncertainty in the Concept dimension of reuse.

6.2 | Trend 2. Reusable system features

The rise of cyber-software engineering is also relevant to software reuse. More and more, machine learning and computational search techniques have already been applied to many problems throughout the software engineering life cycle. This is reshaping the role that new approaches for reusable system feature identification are playing. These approaches are becoming *suggestion engines*. They are called suggestion engines because they do not output a single reusable system feature but a ranking of reusable system features. The software engineer is expected to inspect the ranking and either choose one of the solutions or trigger a different search to produce a new ranking. This is the case of *feature location*.[39] Modern feature location approaches output suggestions that are in turn refined by software engineers. As a result, software engineering effort is moving up the abstraction chain to focus on refining the output or inputs of these suggestion engines. This is an example of improving our ability to handle uncertainty in the Context dimension of reuse.

6.3 | Trend 3. Reusable services

The continuous reengineering of conventional systems into service-based software with different granularity levels (ie, from web services to microservices) is still a popular development technique that builds and deploy noncritical functionality as smaller code assets that can be reused using Internet protocols. Successful examples like Netflix using an MSA and supported by specific platforms like Docker and Kubernetes are paving the way for web developers to achieve better reuse ratios instead of other source code components and easier to deploy. However, compared with other reusable assets, the perfect size of such reusable microservices is not well-defined, as we can find examples of very small functionality engineering as a micro-service. Therefore, it is not so easy to create scalable and reusable microservices when these are used by only one other service.

6.4 | Trend 4. Reusable machine learning components

Technologies such as machine learning are not only helping to stretch the boundaries of retrieval techniques but also beginning to be encapsulated as the primary content of reusable components. An example may be found in the powerful machine vision systems typical of automated vehicles. The problem is not so much in the specification of the components but in their testing. Given the training data and continual self-modification of components based on techniques like deep neural networks, it can be extremely difficult to demonstrate that the component's behavior fully conforms to its specification. The major research challenge in this area is adequate testing of these new kinds of reusable components, which may end up redefining the meaning of specification conformance. This is an example of dealing with uncertainty in the Content dimension of reuse.

7 | CONCLUSION

The results of our practitioner survey have made it abundantly clear that Rubén Prieto-Diaz was right in his prediction that "software reuse will cease to exist": In the 26 years since the original survey was performed, practitioners have learned to integrate software reuse so completely into their development processes that it is no longer even a matter of discussion. But were the predictions right that software reuse would cease to exist as an evolving, research-worthy discipline? Both the vibrant industrial sector reuse communities and the research trends described in this paper indicate that the answer is emphatically No. Our ambition continues to drive us to harness emerging technologies to pursue reuse research in new contexts, with new artifacts, and for new, even more powerful search, retrieval, and development processes. In conclusion, we cite one final prediction, made over 30 years ago by Fred Brooksm,[78] that reuse would remain the primary means of confronting the essential complexity of developing software. The implications of this are clear: No matter where our ambition takes us in the future, uncertain world, reuse will always be with us, and there will always be new opportunities for software reuse.

ACKNOWLEDGMENTS

R. Capilla's work is supported by MCIU-AEI TIN2017-90644-REDT. B.Gallina is supported by EU and VINNOVA via the ECSEL JU under grant agreement No 692474, AMASS project.

ORCID

Rafael Capilla https://orcid.org/0000-0002-6943-1285
Barbara Gallina https://orcid.org/0000-0002-6952-1053

REFERENCES

1. Naur P, Randell B. Software engineering: Report on a conference. In: Nato science committee NATO; 1969; Garmisch, Germany:231.
2. Erdogmus H, Medvidovic N, Paulisch F. 50 years of software engineering. *SOFTW*. 2018;35(5).
3. McIlroy MD. Mass produced software components. Peter Naur and Brian Randell, eds. *Nato science committee NATO*. Belgium: Scientific Affairs Division NATO; 1969:1–136.
4. Bigerstaff TJ, Perlis AJ. *Software reusability. Vol I: concepts and models*. New York: ACM Press; 1989.
5. Latour L, Wheeler T, Frakes W. Descriptive and predictive aspects of the 3cs model: Seta1 working group summary. In: Workshop: Methods and tools for Reuse Syracuse University CASE Center; 1990.
6. Weide B, Ogden WF, Zweben SH. Reusable software components, advances in computers. *Transp Res Rec: J Transp Res Board*. 1991;33:1-65.
7. Jézéquel JM, Meyer B. Design by contract: the lessons of ariane. *COMPUT*. 1997;30(2):129-130.
8. Prieto-Díaz R. Domain analysis for reusability. *Compsac'87 IEEE*; 1987:23-29.

9. Prieto-Díaz R. The dissapearance of software reuse. In: Third International Conference on Software Reuse: Advances in Software Reusability, Rio de Janeiro, Brazil; 1994.
10. Krueger CW. Software reuse. *ACM Computing Surveys*. 1992;24(2):131-183.
11. Prieto-Diaz R, Freeman P. Classifying software for reusability. *IEEE Softw*. 1987;4(1):6-16.
12. Simos MA. Organization domain modeling (ODM): formalizing the core domain modeling life cycle. In: Proceedings of the 1995 Symposium on Software Reusability (SSR) ACM, Seattle, USA; 1995:196-205.
13. Capilla R. Application of domain analysis to knowledge reuse. *8th Workshop on Institutionalizing Software Reuse*; 1996:1-6.
14. Arango GF. Domain engineering for software reuse. *PhD thesis*. Irvine: University of California, California, USA; 1988.
15. Prieto-Diaz R, Arango GF. *Domain analysis and software systems modeling*: IEEE Press; 1991.
16. Frakes W, Prieto-Diaz R, fox C. Dare: domain analysis and reuse environment. *Ann Softw Eng*. 1998;5(1):125-141.
17. Neighbors JM. The draco approach for constructing reusable components. *IEEE Trans Softw Eng*. 1984;10:564-574.
18. Batory D, O'Malley S. The design and implementation of hierarchical software systems with reusable components. *ACM Trans Softw Eng Methodol*. 1992;4:355-398.
19. Czarnecki K, Eisenecher U. *Generative programming: methods, tools, and applications*. Reading, MA, USA: Addison-Wesley; 2000.
20. Bigerstaff TJ, Perlis AJ. *Software reusability*. *Vol II: applications and experience*. New York: ACM Press; 1989.
21. Anderson W. What COTS and software reuse teach us about SOA. In: 6th International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07) IEEE, Banff, Alberta, Canada; 2007:1-6.
22. Barnes B, Durek T, Gaffney J, Pyster A. A framework and economic foundation for software reuse. *Software Reuse: Emering Technology*: IEEE, Los Alamitos, California, USA; 1988:77-78.
23. Gaffney J. E:, Durek TA. Software reuse—key to enhanced productivity: some quantitative models. *Inf Softw Technol*. 1989;31(5):258-267.
24. Favaro J. What price reusability? *First Symposium on Environments and Tools for Ada*. Redondo Beach, California, USA; 1990:115-124.
25. Poulin JS. *Measuring software reuse: principles, practices, and economic models*. Reading, MA, USA: Addison-Wesley Longman; 1996.
26. Lim WC. *Managing software reuse*. New Jersey: Prentice Hall; 1994.
27. Favaro JM, Favaro KE, Favaro PF. Value based software reuse investment. *Ann Softw Eng*. 1998;5(1):5-52.
28. Biffl S, Aurum A, Boehm B, Erdogmus H, Grünbacher P. *Value-based software engineering*. Berlin, Heidelberg: Springer Verlag; 2006.
29. Clements PC, Northrop L. *Software product lines: practices and patterns*, *SEI Series in Software Engineering*: Addison-Wesley; 2002.
30. Ternité T. Process lines: a product line approach designed for process model development. In: 2009 35th Euromicro Conference on Software Engineering and Advanced Applications; 2009; Patras, Greece:173-180.
31. Gallina B, Sljivo I, Jaradat O. Towards a safety-oriented process line for enabling reuse in safety critical systems development and certification. In: 2012 35th Annual IEEE Software Engineering Workshop; 2012:148-157.
32. Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A. A survey of variability modeling in industrial practice. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems VaMoS'13. ACM; 2013; New York, NY, USA:7:1-7:8.
33. Pham NH, Nguyen HA, Nguyen TT, Al-Kofahi JM, Nguyen TN. Complete and accurate clone detection in graph-based models. In: Proceedings of the 31st International Conference on Software engineering ICSE '09. IEEE Computer Society; 2009; Washington, DC, USA:276-286.
34. Assunção WesleyK. G., Lopez-Herrejon RE, Linsbauer L, Vergilio SR, Egyed A. Reengineering legacy applications into software product lines: a systematic mapping. *Empir Softw Eng*. 2017;22(6):2972-016.
35. Martinez J, Assunção WKG, Ziadi T. EPLA: a catalog of extractive spl adoption case studies. In: Proceedings of the 21st International Systems and Software Product Line Conference -Volume B SPLC'17 ACM; 2017; New York, NY, USA:38-41.
36. Round-trip engineering and variability management platform and process (revamp2 itea3project). Identification of relevant state of the art. http://www.revamp2-project.eu/images/documentation/REVAMP2_D3.1_v05.pdf; 2017.
37. Beuche D, Schulze M, Duvigneau M. When 150% is too much: supporting product centric viewpoints in an industrial product line. In: Proceedings of the 20th International Systems and Software Product Line Conference SPLC '16. ACM; 2016; New York, NY, USA:262-269.
38. Martinez J, Ziadi T, Bissyandé TF, Klein J, Le Traon Y. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C); 2017; Buenos Aires, Argentina:67-70.
39. Font J, Arcega L, Haugen O, Cetina C. Achieving feature location in families of models through the use of search-based software engineering. *IEEE Trans Evol Comput*. 2018;22(3):363-377.
40. Font J, Arcega L, Haugen O, Cetina C. Feature location in models through a genetic algorithm driven by information retrieval techniques. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems MODELS '16. ACM; 2016; New York, NY, USA:272-282.
41. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI- 90-TR-21, Carnegie-Mellon University Software Engineering Institute; 1990.
42. Bosch J. *Design and use of software architectures: adopting and evolving a product-line approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co; 2000.
43. Desmet B, Vallejos J, Costanza P, Meuter WD, D'Hondt T. Context-oriented domain analysis. In: 10th International and Interdisciplinary Conference on Modeling and Using Context. Cambridge, UK: Sprnger; 2007:178-191.

44. Capilla R, Ortiz O, Hinchey M. Context variability for context-aware systems. *IEEE Comput*. 2014;47(2):85-87.

45. Mens K, Capilla R, Hartmann H, Kropf T. Modeling and managing context-aware systems' variability. *IEEE Softw*. 2017;34(6):58-63.

46. International Organization for Standardization and International Electrotechnical Commission and IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association. ISO 24765: systems and software engineering - vocabulary; 2017.

47. Gallina Barbara. Towards enabling reuse in the context of safety-critical product lines. In: 5th IEEE/ACM International workshop on product line approaches in software engineering. PLEASE 2015; 2015; Florence, Italy:15-18.

48. International Electrotechnical Comission. IEC 61508: functional safety of electrical/electronic/programmable electronic safety-related systems; 2010.

49. Bishop PG, Bloomfield RE, Froome PKD. Justifying the use of software of uncertain pedigree (soup) in safety-related applications. Report No: CRR336 HSE Books ISBN 0 7176 2010 7; 2001.

50. International Electrotechnical Comission. IEC 62304: medical device software - software life cycle processes; 2006.

51. Radio Technical Commission for Aeronautics (RTCA). DO-297 -integrated modular avionics (IMA) development guidance and certification considerations; 2005.

52. the Open Group. Future airborne capability environment (FACE) reference architecture; 2012.

53. International Organization for Standardization. ISO 26262: Road vehicles — Functional safety; 2011.

54. AUTOSAR Website. AUTOSAR (AUTomotive Open System ARchitecture). http://www.autosar.org

55. AUTOSAR Website. Specification of safety extensions AUTOSAR CP release 4.4.0). https://www.autosar.org/standards/classic-platform/classic-platform-440/

56. European cooperation for space standardization. ECSS-Q-HB-80-01A: Space product assurance - Reuse of existing software; 2011.

57. Hardung B, Kölzow T, Krüger A. Reuse of software in distributed embedded automotive systems. In: Fourth ACM International Conference on Embedded Software (emsoft 2004) September 27-29, 2004, Pisa; 2004; Italy ACM:203-210.

58. Chrisofakis E, Junghanns A, Kehrer C, Rink A. Simulation-based development of automotive control software with modelica. In: 8th International Modelica Conference Linköping University Electronic Press; 2011; Dresden, Germany:1-7.

59. Nešić D, Nyberg M, Gallina B. Constructing product-line safety cases from contract-based specifications. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing SAC '19 ACM; 2019; New York, NY, USA:2022-2031. http://doi.acm.org/10.1145/3297280.3297479

60. Favaro J, Savoia G, D'Allessandro M. Adabeans: an architecture for component-based development of aerospace applications, using uml as a visual composition language. DASIA-Data Systems in Aerospace, 25-28 May, 1998, Athens, Greece Edited by B. KaldeichSchürmann. ESA SP-422. Paris, European Space Agency.

61. Panunzio M, Vardanega T. A component-based process with separation of concerns for the development of embedded real-time software systems. *J Syst Softw*. 2014;96:105-121.

62. AMASS Consortium. AMASS (Architecture-driven, multi-concern and seamless assurance and certification of cyber-physical systems). (Last accessed: July 3, 2019). http://www.amass-ecsel.eu.

63. Javed MA, Gallina B. Safety-oriented process line engineering via seamless integration between EPF composer and BVR tool. In: Proceedings of the 22Nd International Systems and Software Product Line Conference (SPLC) - Volume 2. ACM Digital Library; 2018; Gothenburg, Sweden, New York, NY, USA:23-28.

64. Javed MA, Gallina B, Carlsson A. Towards variant management and change impact analysis in safety-oriented process-product lines. In: Proceedings of the 34thACM/SIGAPP Symposium on Applied Computing SAC'19. ACM; 2019; New York, NY, USA:2372-2375. http://doi.acm.org/10.1145/3297280.3297634

65. Gallina B. Quantitative evaluation of tailoring within spice-compliant security-informed safety-oriented process lines. *J Soft Evol Process*. 2019.

66. The Consultative Committee for Space Data Systems (CCSDS). Report concerning space data system standards. spacecraft onboard interface services, INFORMATIONAL REPORT CCSDS 850.0-G-2, GREEN BOOK, Washington DC, USA; 2013.

67. EEIG ERTMS Users Group. ERTMS/ETCS RAMS requirements specification: chapter 2 - RAM. http://docplayer.net/20959132-Ertms-etcs-rams-requirements-specification-chapter-2-ram.html

68. Shift2Rail Joint Undertaking. Multi-annual action plan. https://shift2rail.org/

69. ter Beek MH, Fantechi A, Gnesi S. Product line models of large cyber-physical systems: the case of ERTMS/ETCS. In: Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018; Spring September 10; Gothenburg, Sweden:208-214.

70. Jakovljevic M, Geven A, Simanic-John N, Saatci DM. Next-gen train control / management (TCMS) architectures: drive-by-data system integration approach. In: Proceeedings of the 9th European Congress on Embedded Real Time Software and Systems ERTS Toulouse; 2018; France.

71. Font J, Arcega L, Haugen O, Cetina C. Building software product lines from conceptualized model patterns. In: Proceedings of the 19th International Conference on Software Product Line SPLC '15. ACM; 2015; New York, NY, USA:46-55.

72. Marcén AC, Pérez F, Cetina C. Ontological evolutionary encoding to bridge machine learning and conceptual models. *Approach and industrial evaluation*. Switzerland; 2017:491-505.

73. Pérez F, Font J, Arcega L, Cetina C. Collaborative feature location in models through automatic query expansion. *Autom Softw Eng*. 2019;26(1).

74. Frakes WB, Fox CJ. Sixteen questions about software reuse. *Commun ACM*. 1993;38(6):75-87.

75. Duggan W. *Creative strategy: a guide for innovation*. New York: Columbia University Press; 2012.

76. Favaro J. Knowledge reuse in innovation. *International Conference on Knowledge Discovery, IC3k*; 2015:444-456.

77. Favaro J, Mazzini S, Schreiner R, de Koning H, Olive X. Next generation requirements engineering. *Proceedings Incose International Symposium*. Rome, Italy; 2012:461-474.

78. Brooks FPJr. No silver bullet essence and accidents of software engineering. *Comput*. 1987;20(04):10-19.

How to cite this article: Capilla R, Gallina B, Cetina C, Favaro J. Opportunities for software reuse in an uncertain world: From past to emerging trends. *J Softw Evol Proc*. 2019;31:e2217. https://doi.org/10.1002/smr.2217