

Compilation of Distributed Programs to Services Using Multiple Programming Languages

Thomas M. Prinz^a

Course Evaluation Service, Friedrich Schiller University Jena, Jena, Germany

Keywords: Service, Compiler, Distributed Programming, Programming Language, Engineering.

Abstract: Service-orientation recommends dividing software into separate independent services, with each service being implemented in the programming language that best fits into the service's problem space. However, data must be shared between the distributed services, so common data models and interfaces must be defined in each programming language used. This leads to a higher development effort and dependencies, while neglecting the benefits. This paper explains a new idea that arranges a distributed program as if it is a single one, even though it consists of different parts using possibly different programming languages. For this purpose, the idea of meta network programming languages is introduced. They are based on network machines and hide the complexity arising during development of distributed software. A compiler translates and distributes these programs by splitting them into several parts. As a result, this should reduce the overhead of developing distributed general purpose software. The intention of this position paper is to give new ideas to implement distributed programs in the future. An implementation of the idea does not exist yet.


1 INTRODUCTION

The development and implementation of software is a complex process. To deal with this complexity, researchers and practitioners have defined different strategies. One strategy is to divide the software into different modules, whereby each module can be developed separately from the others. To reduce functional implementation effort, each module should be implemented in the programming language and tool stack that best fits its problem space. Emerging technologies like service-oriented architectures (SOA) make this possible. If services are very closed (atomic) in their functionality, they are called microservices. A microservice communicates over a network, is deployed independently from other services, and uses the programming language and tool stack that best fits its needs (Newman, 2015). For example, if someone needs to develop a software, which deals with objects, but also with machine learning, parts can be implemented in Java, a typical object-oriented language, and in Python, known for its data science packages.

When software is divided into different subsystems, the subsystems must communicate with each

other to achieve the overall goal of the software. Therefore, they need communication interfaces. In software development with one programming language, these interfaces are realized with function (method) declarations. When using multiple programming languages, however, not all functions are located in the same execution environment; they could be anywhere in the network or on the machine and are called via network or other approaches.

The big difference between calls within a programming language or outside via services is mainly in their support (De Paoli, 2018). In monolingual programming, most of the program code is known at compile time, so most of the code is known during development; integrated development environments (IDEs) can use these information and, therefore, can fully support the development. Modern IDEs identify, among other things, calls to undefined methods, access violations, and incorrectly placed method parameters. In those cases, the IDE provides immediate feedback during coding and, usually, some error handling. Furthermore, since the software is developed in a single language, compilers can translate it into (virtual) machine code and — in most cases and by defining the execution environment — they are immediately executable. On the contrary, in the context of service-based and distributed software architectures,

^a  <https://orcid.org/0000-0001-9602-3482>

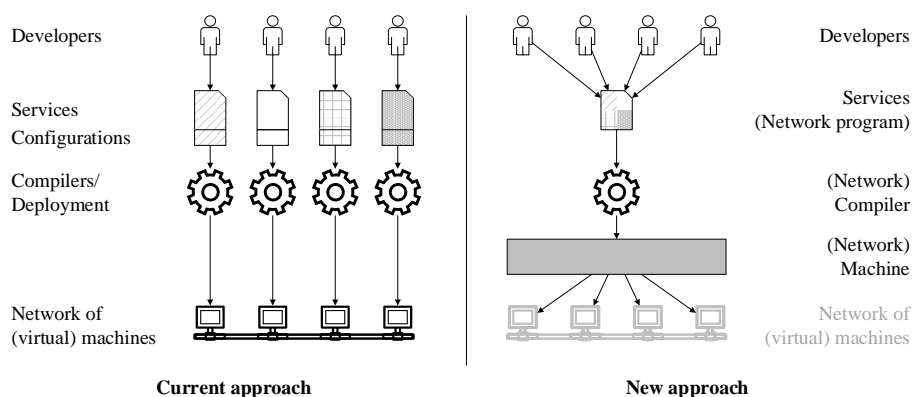


Figure 1: The current approach to develop distributed and service-oriented software (left) and our new approach (right).

such IDE support is weak. The reason is that the IDE only knows one service and not the overall software architecture. Developers need to control each service and verify that they meet predefined specifications. Without this quality check, services will not be called properly — even if they are implemented correctly.

Besides the described disadvantages in IDE support, the current practice has further disadvantages resulting in a high development effort. The shared data models between services must be implemented in all programming languages used, which leads to a high and unnecessary programming overhead. In addition, objects shared between services need serializers and deserializers and Application Programming Interfaces (APIs) must be implemented for services as well as for calls to them. (Apel et al., 2019) have shown in a real project that the ratio of implementation overhead is sometimes less than or equal to 1 : 3 between functional and additional code, i. e., for 100 lines of functional code 300 lines of organizational code are required. In general, *the current approach leads to a high workload during programming, development, specification, and maintenance and, therefore, to high costs and time expenditure.* Especially, for small development teams, these cost and time factors become crucial.

Figure 1 (left) visualizes the current approach of distributed software development. Developers define, implement, and deploy services separately, which eventually run on different virtual machines and realize the software.

Figure 1 (right) illustrates our new idea using a compiler for distributed and service-oriented software. At first, we reduce common properties of distributed, multiprocessor, and service-oriented systems to a single, abstract base which we call a *network machine*. A network machine hides the complexity of the network of (virtual) machines to simplify imagination of the execution environment during development. In

addition, it organizes sharing of data, message exchange, and calls of distributed functionality (e. g., service calls). Thus, developers can imagine, their program runs on a single machine, where researchers claim that it is easier to think about and improves code quality (Kozlovics, 2019). In ease, network machines describe runtime environments. These environments execute so-called *network programs*. Network programs describe the combination of all distributed parts of a distributed software but hiding its complexity by ordinary method and function calls. The advantage of a network program is the ability to implement different functions of the software in different programming languages. It is the task of a *network compiler* to split the program into separate modules and translate each module into a different language with its own (shared) execution environment.

Advantages of our idea would be that IDEs can support developers across service and language boundaries and, when good algorithms are found to split programs automatically in such a way that it has the best performance neglecting communication overhead, the overall program performance can be improved. Furthermore, programming, development, and specification would become easier and faster as with conventional approaches, and, by the way, development becomes also easier when using different programming languages in a non-distributed context. This paper introduces the idea and basic concepts while there is no concrete implementation yet.

This paper is structured as follows: Section 2 considers related work in distributed systems and their implementation. Section 3 starts to generalize the approach and defines network machines running network programs defined in Section 4. Subsequently, Section 5 describes a compiler architecture for network programs. Section 6 concludes and briefly discusses the idea.

2 RELATED WORK

The idea of implementing software in a unified language is not new and came up early in research and industry. For this reason, many approaches for languages for data models and functionality have evolved in the past. For example, the *Unified Modeling Language* (UML) (Object Management Group, 2017b) is such a unified language, in which the software development community and industry use various diagrams that are widely used as modeling language standard (Oquendo et al., 2016) (especially, class, use case, and sequence diagrams (Dobing and Parsons, 2006)). The *Action Language for Foundational UML* (Alf) (Object Management Group, 2017a) and the *Semantics of a Foundational Subset for Executable UML Models* (fUML) (Object Management Group, 2018) are standards for describing complete systems in UML down to the behavior of individual functions, but their application is not trivial and requires scientific knowledge (Ciccozzi et al., 2019). In contrary, our idea would be similar to usual software development and should be easier to learn.

Interoperability of different systems, programming languages, and platforms is the goal of the *Common Object Request Broker Architecture* (CORBA) (Object Management Group, 2021). CORBA follows a strong object-oriented approach and considers each functionality to be bounded in (remote) objects as methods. Developers describe the signature of such methods in an *Interface Definition Language* (IDL), compile them into programming language specific interface stubs, and add their functionality. An *Object Request Broker* (ORB) handles requests to the methods and delegate them to specific implementations regardless where they are and how they are implemented. Although CORBA has similarities with our approach, it also differs. CORBA focuses on interacting objects; our approach, however, handles objects just as complex data structures and provide functions/methods as (micro)services. Furthermore, CORBA defines function signatures separately in the IDL, compiles them into stubs, and force to implement them subsequently; however, our approach allows to integrate both the implementation and the “cross-language” signature. Finally and importantly, CORBA is expensive for companies as ORBs are complex and, therefore, usually are licensed from commercial vendors; our approach, however, should result in a less complex environment and, therefore, should be more attractive for small companies.

Alternative approaches try to minimize the effort between design and implementation. For example, *Swagger* (Swagger, 2023a) allows the specification

of REST APIs in an *OpenAPI* specification (Swagger, 2023b) and the generation of client and server code skeletons. *JHipster* (Raible, 2018) generates complete applications within a predefined technology stack. Both Swagger and JHipster are limited by ignoring the system behavior. The framework *Flutter* (Google, 2023) describes applications in *Dart* and compiles them into native cross-platform applications, but does not translate parts of the program into individual target machines and languages. Some web template and behavioral languages, e. g., *pug* (pug, 2023), *Haml* (Catlin, 2023), and *GWT* (Tacy et al., 2013), allow software descriptions and compile them into web applications. They are powerful but limited to web applications, JavaScript, and Java.

Although limited to C++, C, and Fortran, *OpenMP* (OpenMP Architecture Review Board, 2018) has similarities to our ideas and has become a standard for programming shared memory systems in High Performance Computing (HPC). Rather than focusing on multiprocessors on the same machine as *OpenCL* (Khronos OpenCL Working Group, 2023), it enables high-level parallel programming and portability (Yu et al., 2020) over a network. Programs are translated into heterogeneous multiprocessor systems, but this complexity is hidden from the programmer. After compilation, the OpenMP environment automatically moves different machine code to the processors and handles data exchange between them. Our idea is similar by hiding details about the infrastructure. However, our focus is not on HPC, but on programming for general purpose.

The game development engine *Unity* (Unity Technologies, 2023) follows a similar idea like ours. It allows to describe games in a subset of C# running on Mono (Mono Project, 2023), an open-source .NET framework. Mono is based on the Common Language Infrastructure (CLI) standard (Ecma International, 2012), which allows programs to run on different operating systems. The use of C# as a de-facto meta programming language follows our idea to define data models and services independently of programming languages. However, instead of translating the models and services into a single language and runtime environment, our approach is to distribute them across different languages and environments. Another example in game development is the *GameEngine* of (Apel, 2018) for the definition of massively multiplayer online games (MMOG). The novelty of his approach was the automatic generation of code for communication, controllers, etc. as well as their compilation at runtime. This made games leaner in terms of shortened implementation times and lines of code. However, its focus on MMOGs makes it dif-

difficult for use in general development.

Instead of simplifying the development of software with a specific execution environment as introduced in the previous examples, the *web computer* hides the complexity during the development and execution with an own operating system (Kozlovics, 2019). It addresses similar problems in software development as we do, focusing on the assumption that the software should be implemented for a single computer, a single user, and as single executed program. Its main approach is to hide all network and memory sharing specific issues from the developer by distributing memory changes across all physical devices and automatically creating network communications. The approach seems promising, but it is still unclear how web computer applications are developed and compiled. It is also limited to web applications.

There are only a few papers that try to compile complete service systems, examples are Singer et al. (Singer, 2016; Geisriegler et al., 2017) and Prinz et al. (Prinz et al., 2014; Prinz et al., 2015). However, both research groups have a background in business process management and focus on business processes. However, current process modeling languages are not able to describe the full functionality of general purpose software.

3 NETWORK MACHINES

Business processes, HPC, distributed, and service-oriented software have their differences in implementation and execution. However, they also have strong similarities: They are implemented to run on a network of possibly different (virtual) machines, which may have their own supported programming languages and tool stacks. From an abstract perspective, in multiprocessor systems such as those used in HPC, each multiprocessor can perform a different function that can be executed on one machine with a different instruction set than on the other machines. In SOA, services are the interfaces of functions, where the service could be implemented in different programming languages, run on different machines, and have its own or a shared and distributed memory. Because of the strong similarities between them, our idea is to combine them into a common ground of distributed software that we call a *network machine*.

A network machine is a lightweight runtime environment for distributed software. From the developer's point of view, a network machine executes a *network program* that is defined in detail in the next section. The final compiled files of a network program are *containers*, which contain various *services*,

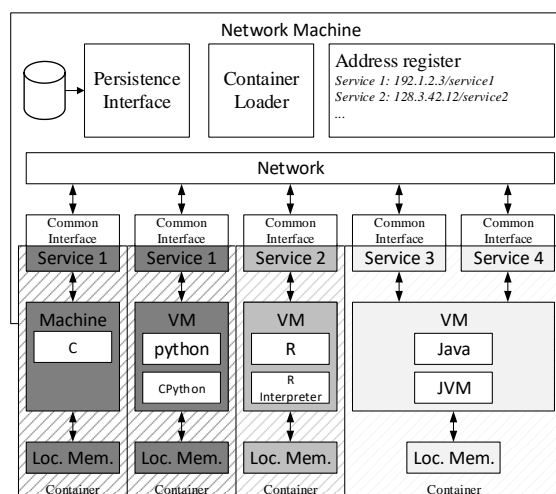


Figure 2: A network machine with containers that define services, (virtual) machines (devices), used programming languages, and execution environments with local and distributed memories. In addition, the network machine provides a container loader, a persistence interface, a common interface for services, and an address register where it stores the network addresses of the services.

but all services within a container are compiled from the same programming language, e. g., there could be containers for Java, R, C, and Python services. Basically, one technology that can be used and imagined for containers is Docker (Docker Inc., 2023) and Kubernetes (Burns et al., 2019). The network machine loads these containers and makes them available. At this moment, the software is running and accessible. Since the network machine knows all services, if one service calls another, the machine can redirect the call to the right service. Figure 2 illustrates the conceptual idea of network machines.

From a technical point of view, a network machine hides the complexity of a set of different, possibly virtual, computing machines (devices) connected to each other over a network (as shown in Figure 2). The network serves as bus for the exchange of data and messages between the devices. Each device hosts a container and, therefore, provides all the container's services. Since the network machine knows the network, it can assign specific addresses to each service (like memory addresses on normal machines) and store them in an *address register*. All services can be accessed via a *common interface*, which processes data input and output. This is comparable to the handling of functions in machine languages: The common interface deserializes each input parameter and puts the result into the service context so that the service can access it. Then the interface calls the service functionality. The results of the service call are serialized by the interface and sent back to the caller.

The service functionality is executed in a separate (possibly virtual) environment. These are actual runtime environments for the programming languages used in the services, e. g., Java Virtual Machines (JVM) (Lindholm et al., 2014) or Python and R runtime environments. Since a network machine is also a runtime environment, behind a service can again be an own network machine. In addition, each service has its own local memory and shares data only via the interfaces. Global data should be used with caution, as with any distributed software. Network machines provide global data via a *persistence interface* for all data structures defined in the network program and shared by the services. This persistence interface allows easy and continuous storage of data, its access, modification, and deletion. Since services could run in parallel when asynchronous service calls are used, this interface solves the race condition between parallel accesses, e. g., by locking mechanisms.

4 NETWORK PROGRAMS

Programming distributed software is a challenge because the developer cannot simply call another function in a distributed part of the program, but must call an external service using the right parameters, format, etc. It is helpful during development to imagine that all functions of a program are located in the same environment, regardless of where they are actually located (Kozlovics, 2019). The idea of *network programs* creates this situation. Network programs are executed on the previously defined network machines and are written in a so-called *network programming language* that mainly defines data models and the software structure as services. The advantages of such network programs are: (1) they define the interfaces of all services, (2) service calls are more similar to function/method calls, and (3) they describe shared data models only once.

Figure 3 shows an example of a small network program. It is written in a Java-like language only as an example. At the beginning, the program defines the data model (class) `Pair` with two fields `a` and `b` and a constructor. It also defines a class `Computation`, which contains two methods. The method `handlePairs` is written in Java and transfers a nested integer array of pairs into a list of objects of type `Pair`. At the end, the method returns the sum of the list of pairs by calling the `computeSums` method, which is written in R. The method `computeSums` requires an array (or list) of pairs defined by the class `Pair`. It then calculates the sum of the two fields `a` and `b` of each `Pair` object and returns it as an array.

```

1 class Pair {
2   public int a, b;
3   Pair(int a, int b) {
4     this.a = a; this.b = b;
5   }
6 }

1 class Computation {
2   @Java
3   public int[] handlePairs(int[][] pairs) {
4     Pair[] pairList = new Pair[pairs.length];
5     for (int i = 0; i < pairs.length; i++) {
6       int a = pairs[i][0], b = pairs[i][1];
7       pairList[i] = new Pair(a, b);
8     }
9     return this.computeSums(pairList);
10  }
11  @R
12  public int[] computeSums(Pair[] pairs) {
13    sapply(pairs, function(pair) {
14      pair$a + pair$b
15    })
16  }
17 }

```

Figure 3: An example of a network program.

The implementation of the program with the current state-of-the-art is not a challenge but time consuming. First the data structure `Pair` must be implemented in Java and in R. Then, the Java part of the program must be implemented and made available as a service, e. g., with REST. In addition, developers must implement the R part of the program and also make it available as a service. Since data is transferred, a protocol must be defined or an existing one like JSON (Ecma International, 2017) must be used for the exchange. Furthermore, the data must be mapped to the correct parameters.

The implementation overhead can cause developers to implement the program/software in a single language instead of multiple ones. However, some problems can be solved, computed, and implemented much more efficient in other programming languages. This advantage is then missing and, therefore, leads to increasing implementation costs (Apel et al., 2019). Network programs should improve this situation since developers avoid the previous described steps.

5 COMPILER ARCHITECTURE FOR NETWORK PROGRAMS

Network programs are transferred to a runnable software with the help of a compiler. Basically, the architecture of such a compiler can follow a similar structure as classical compilers (e. g., (Cooper and Torc-

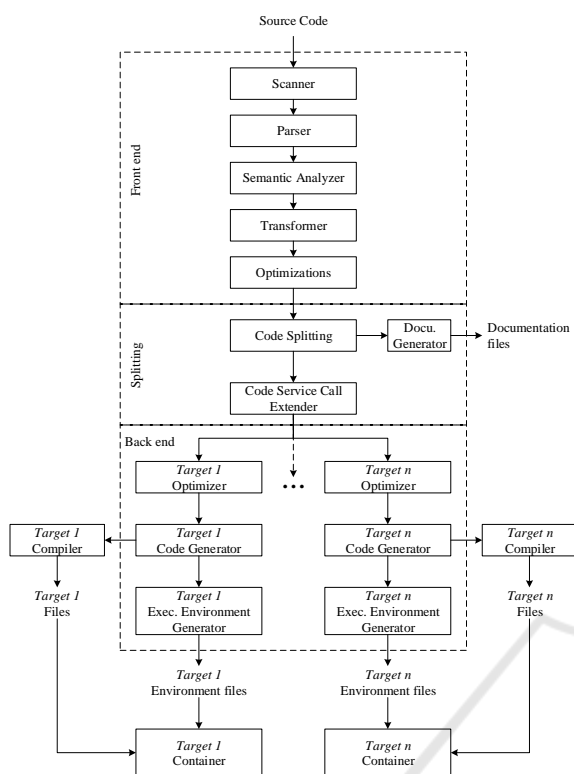


Figure 4: The compiler architecture for network programs.

zon, 2011)). It consists of a *front end*, which processes the source program (the input) and translates it into an internal *intermediate representation* (IR), and a *back end* that translates it into the output language. Although so-called *cross* compilers are capable of generating code for different output languages, they produce code for the entire program in one language. Our idea of a compiler architecture overcomes this limitation and allows to split the source program into parts and compile each part into a different programming language. Furthermore, it generates additional code that allows the parts to communicate with each other (i. e., method/function calls). The splitting itself is part of the *optimizations* of the compiler.

Figure 4 shows the phases of the compiler architecture we propose for network programs. The front end of the compiler has the same structure as in the classical compiler architecture. First, the *scanner* produces a stream of tokens from the source program, which is used in the *parser* to check the grammar of the programs and build the *Abstract Syntax Tree* (AST). The *semantic analyzer* executes various algorithms to find errors in the program as early as possible, e. g., (static) type violations, access violations, undefined variables, and so on. The compiler translates the confirmed AST into a an internal representation — the IR — in the *transformer* to make some

optimizations easier to apply to it. All optimizations are independent with respect to the target code(s).

At this point, it is important to mention that the front end should mainly process those parts of a network program that are written in a network programming language. Code which is already written in the target languages is not checked in the front end. For them, it uses the language-specific compilers and provides the whole context (e. g., data models and method schemes with arguments) in the back end.

After the front end, the compiler architecture has a *splitting* section, comparable to an optimization. The compiler tries to split the IR into a distributed IR, where each distributed part of the IR can be translated into a different target language and machine. The compiler takes the code that is already available in all target languages and adds translations of data models to them if necessary. For example, the network program in Figure 3 abstractly describes the class *Pair*. Since this class is needed in the Java and R sections of the program, this class should be available in both Java and R. For this reason, the compiler translates the abstract definitions of the classes into programming language specific definitions in both languages.

Since splitting the code into different parts leads to a more complex software architecture, the resulting architecture should be well documented. The compiler should provide a *Documentation Generator* phase, which generates both an architectural description and information about defined data models, e. g., in the form of class diagrams.

After code splitting and assignment of target programming languages, the compiler extends the different parts with additional code in the IR. The *Code Service Call Extender* wraps each part with a service. In other words, they can be called by other program parts with the common interface explained for the network machines. Therefore, the extender also has to add code for serialization and deserialization of data. For example, most modern programming languages have libraries allowing to define RESTful service interfaces (Fielding and Taylor, 2002) and to import and export complex data models as JSON (Ecma International, 2017) or CSV (Shafranovich, 2005). These implementations can be used.

Following the code extender compiler phase, the back end translates the program parts step by step in the target programming language. Since this translation is language dependent, the compiler can perform some optimizations in additional *Optimizer* phases. For the optimized partial code, a language dependent *Code Generator* produces high-level code in the target programming language. Compiling the partial program itself is outside the compiler’s scope — for

most programming languages there are compilers that generate executable files.

Many modern programming languages, such as Java, R, Python, and JavaScript, require special environments to be executable. Fortunately, there are tools to create such environments (e. g., (Docker Inc., 2023) and (Burns et al., 2019)). One goal of the compiler is to generate containers for each partial program describing their execution environments and the services. This is done in the *Execution Environment Generator* and bundled in containers. As a result, the compiler generates a complete distributable and executable network program.

6 CONCLUSION / DISCUSSION

This paper presented a new idea for developing distributed programs. An implementation would allow to arrange each distributed program as if it is an undistributed one, although it consists of different parts, no matter which programming language they use. A compiler distributes the program by splitting it into several parts in different programming languages. To achieve this, the program must be defined in a so-called network programming language. A network program defines data models and services, where the service functionality is described in an individually available programming language chosen by the developer. That language that best solves the functionality can be used. The overall concept is based on principles of network machines, which generalize architectures of distributed and multiprocessor systems and are a runtime environment for network programs. In this paper only the idea of network programs and a conceptual compiler architecture are described. It shall give a new view on developing distributed systems and state-of-the-art approaches. Therefore, an implementation does not exist yet.

Of course, the proposed approach does not have only advantages. If development teams do not implement services separately, the resulting services (and their interfaces) might be less reusable and more coupled. This problem depends more on architecture and design decisions than on implementation decisions, which are the focus of our approach. Furthermore, our approach leads to new middleware (the network machine) that must be installed and maintained by enterprises, although the goal is to keep this to a minimum. Finally, some developers specialize in a small number of programming languages. For this reason, such developers might have problems writing, reading, and understanding network programs.

The implementation effort of our approach de-

pends on the number and types of programming languages initially supported. The network programming language itself should be less complex, as should its compiler. One difficulty will be recognizing calls to distributed functions in the target programming languages. The use of a (language-dependent) library could be a solution for a first prototype. The network machine should be the most complex part, as it provides several runtime components. A first prototype should reuse an existing technology stack to keep the implementation simple and feasible. However, compared to other approaches, such as CORBA, the implementation effort and the hurdle to use our approach should be lower.

Future work includes the definition of a first grammar of a network programming language, which allows to include existing languages in the description of functionality. The next step defines a parser of this language and the phase of code extensions for service calls in the compiler. Fortunately, the back end of the compiler is lightweight, since it can reuse all existing compilers for the programming languages used in the program. All this future work contains challenges, but could be feasible in a few years.

REFERENCES

- Apel, S. (2018). Reducing Development Overheads with a Generic and Model-Centric Architecture for Online Games. In *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 21–28. IEEE Computer Society.
- Apel, S., Hertrampf, F., and Späthe, S. (2019). Towards a Metrics-Based Software Quality Rating for a Microservice Architecture - Case Study for a Measurement and Processing Infrastructure. In Lüke, K., Eichler, G., Erfurth, C., and Fahrnberger, G., editors, *Innovations for Community Services - 19th International Conference, I4CS 2019, Wolfsburg, Germany, June 24-26, 2019, Proceedings*, volume 1041 of *Communications in Computer and Information Science*, pages 205–220. Springer.
- Burns, B., Beda, J., and Hightower, K. (2019). *Kubernetes: Up and Running — Dive into the Future of Infrastructure*. O'Reilly, California, USA, 2 edition.
- Catlin, H. (2023). *Haml*. <http://haml.info/>. Last visit in September 2023.
- Ciccozzi, F., Malavolta, I., and Selic, B. (2019). Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, 18(3):2313–2360.
- Cooper, K. D. and Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann, USA, 2nd edition.
- De Paoli, F. (2018). Challenges in services research: A software architecture perspective. In Lazovik, A. and

- Schulte, S., editors, *Advances in Service-Oriented and Cloud Computing*, pages 219–227, Cham. Springer International Publishing.
- Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–113.
- Docker Inc. (2023). Empowering App Development for Developers — Docker. <https://www.docker.com/>. Last visit in September 2023.
- Ecma International (2012). Standard ECMA-335: Common Language Infrastructure (CLI) — 6th edition (June 2012). <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf>.
- Ecma International (2017). Standard ECMA-404: The JSON Data Interchange Syntax — 2nd edition (December 2017). <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.
- Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Trans. Internet Techn.*, 2(2):115–150.
- Geisriegler, M., Kolodiy, M., Stani, S., and Singer, R. (2017). Actor based business process modeling and execution: A reference implementation based on ontology models and microservices. In *43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, Vienna, Austria, August 30 - Sept. 1, 2017*, pages 359–362. IEEE Computer Society.
- Google (2023). Flutter - Beautiful native apps in record time. <https://flutter.dev/>. Last visit in September 2023.
- Khronos OpenCL Working Group (2023). The OpenCL™ Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.
- Kozlovics, S. (2019). The web computer and its operating system: A new approach for creating web applications. In Bozzon, A., Mayo, F. J. D., and Filipe, J., editors, *Proceedings of the 15th International Conference on Web Information Systems and Technologies, WEBIST 2019, Vienna, Austria, September 18-20, 2019*, pages 46–57. ScitePress.
- Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2014). *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, California, USA, 8 edition.
- Mono Project (2023). Mono — Cross platform, open source .NET framework. <https://www.mono-project.com/>. Last visit in September 2023.
- Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O’Reilly, California, USA, 1 edition.
- Object Management Group (2017a). Action Language for Foundational UML (Alf). Concrete Syntax for a UML Action Language, Version 1.1. <https://www.omg.org/spec/ALF/1.1>.
- Object Management Group (2017b). OMG Unified Modeling Language — version 2.5.1. <https://www.omg.org/spec/UML/2.5.1>.
- Object Management Group (2018). Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.4. <https://www.omg.org/spec/ALF/1.1>.
- Object Management Group (2021). Common Object Request Broker Architecture Specification, Version 3.4. <https://www.omg.org/spec/CORBA/3.4>.
- OpenMP Architecture Review Board (2018). OpenMP Application Programming Interface — version 5.0. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>.
- Oquendo, F., Leite, J., and Batista, T. (2016). Executing Software Architecture Descriptions with SysADL. In Tekinerdogan, B., Zdun, U., and Babar, A., editors, *Software Architecture*, pages 129–137, Cham. Springer International Publishing.
- Prinz, T. M., Heinze, T. S., Amme, W., Kretzschmar, J., and Beckstein, C. (2015). Towards a Compiler for Business Processes - A Research Agenda. In de Barros, M. and Rückemann, C.-P., editors, *SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing, Nice, France, March 22–27, 2015. Proceedings*, pages 49–54.
- Prinz, T. M., Spieß, N., and Amme, W. (2014). A first step towards a compiler for business processes. In Cohen, A., editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8409 of *Lecture Notes in Computer Science*, pages 238–243. Springer.
- pug (2023). Getting Started - Pug. <https://pugjs.org/>. Last visit in September 2023.
- Raible, M. (2018). *The JHipster mini-book*. C4Media, USA, 5.0.1 edition.
- Shafraanovich, Y. (2005). RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files. <https://datatracker.ietf.org/doc/html/rfc4180>.
- Singer, R. (2016). Business process modeling and execution - A compiler for distributed microservices. *CoRR*, abs/1601.05976.
- Swagger (2023a). The Best APIs are Built with Swagger Tools. <https://www.swagger.io/>. Last visit in September 2023.
- Swagger (2023b). OpenAPI Specification. <https://swagger.io/specification/>. Last visit in September 2023.
- Tacy, A., Hanson, R., Essington, J., and Tokke, A. (2013). *GWT in Action*. Manning Publications Co., Greenwich, CT, USA, 2nd edition.
- Unity Technologies (2023). Unity — Game Engine. <https://unity.com/>. Last visit in September 2023.
- Yu, C., Royuela, S., and Quiñones, E. (2020). OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. In Stuijk, S. and Corporaal, H., editors, *SCOPES ’20: 23rd International Workshop on Software and Compilers for Embedded Systems, St. Goar, Germany, May 25-26, 2020*, pages 42–47. ACM.