The Tetrapod Model for Knowledge in Computer Science

Michael Kohlhase[a] and Florian Rabe[a]

a FAU Erlangen-Nürnberg, Computer Science

Abstract We present a holistic conceptual framework for knowledge in computer science that classifies languages, tools, and models into five aspects, which we call specification, implementation, exemplification, deduction, and documentation. Humans can naturally and efficiently integrate and move between these five aspects, whereas integrating them is a difficult challenge for software systems, as evidenced by the observation that most existing languages and tools focus on just one of the five aspects. We analyze the complementary strengths and weaknesses of these aspects and discuss the current state and future potential of building multi-aspect systems.

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Social Coding, General-purpose programming

1 Introduction

We introduce the issues motivating this article in three anecdotes from recent practical software engineering.

Anecdote 1 In one of the most high-profile industrial applications of formal methods to date [AWS25], Amazon Web Services recently announced the successful replacement of their central authorization engine (which allows/denies $> 10^9$ AWS requests per second!) with a new, formally verified one. They credit their success to a development cycle that integrated formal specification, implementation, verification, compilation, and testing. For example, to be confident enough to deploy the code, they spent over a year formalizing, documenting, and fuzzing the specification before moving on to the implementation, and eventually ran over a quadrillion test cases on the generated executable code even after its sources had already been formally verified.

Anecdote 2 A colleague of ours shared with us a story about how they were hired to verify a major piece of software against its specification. They were skilled enough to successfully deliver the formal proof without even reading the documentation or analyzing the specification. But later it was discovered that a major bug in the specification was mirrored by a corresponding bug in the implementation and thus went unnoticed despite formal verification.

Anecdote 3 After 10 years of on-and-off working on a port of a major piece of software, specifically the LaTeXML tool [Gin+11], from Perl to Rust, our colleague Deyan Ginev finished the remaining 2/3 of the port in 3 weeks by running Claude Opus, with now 99.5% of arxiv articles being converted to HTML without errors—an achievement that seemed out of reach until recently. This experience is representative of how LLM usage has enabled software development all over the industry that produces code much faster than any human developer can even read it.

Challenge Anecdote 1 is a good example of how modern software engineering increasingly relies on highly evolved computer-support that combines processes like *specification*, *implementation*, *testing*, *documentation*, and *verification* to ensure both quality and efficiency of software development. Anecdote 2 shows how a holistic view on these software engineering aspects is needed that uses all aspects to guard each other, e.g., by extensively unit-testing a formal specification in addition to employing formal verification. This issue becomes ever more pressing as project and team sizes grow and increasingly rely on asynchronous communication between specialists. Anecdote 3 shows that this new perspective is critical for ensuring software correctness: the software of the future, including not just implementations but also specifications, documentation, tests, and proofs, will increasingly be written by AI systems, and the role of humans will shift towards high-level design and correctness assurance.

Despite a general awareness in the field of both the need to tightly integrate these aspects of software engineering and the difficulty to do so, we often observe that

current paradigms, languages, and tools attack them in isolation. This is not too surprising for two reasons.

Firstly, the integration task is formidable, and both theoretical and applied research usually has to restrict attention to very narrowly defined problems to obtain tangible results in realistic time frames. This has led to a diverging multiplicity of specialized university courses, academic conferences, and commercial or free support tools and methods that focus on individual aspects of the software engineering process. For instance, most programming languages are not at the same time ideal for modeling specifications and verifying implementations; unit tests are conceptually simple input/output datasets, but are usually written and maintained as a part of the source code because there is no sufficient integration between the programming language and database technology; and documentation is notoriously difficult to keep tied to and in sync with any formal model or implementation.

Secondly, this separation of aspects has significant advantages. Experience in the field shows that the various aspects have very different constraints and goals, and isolating them allows for powerful optimizations, which are often critical to build scalable tool support. Indeed, it is unclear if a single holistic system that integrates all aspects would be able to capture their complementary advantages — or if it would end up under-performing at each aspect when compared to optimized aspect-specific tools.

Contribution This paper aims to emphasize the interconnectedness of the aspects and pose the question how systematically holistic processes and tools could be built in the future and how they would impact software engineering.

Concretely, we discuss the design space for multi-aspect software engineering languages and tools in more detail: We adapt the **tetrapod model** [Car+21] — which was originally devised as a holistic conceptualization of *mathematical* knowledge — to software engineering (see Figure 1). Table 1 gives an overview of our classification of aspects.

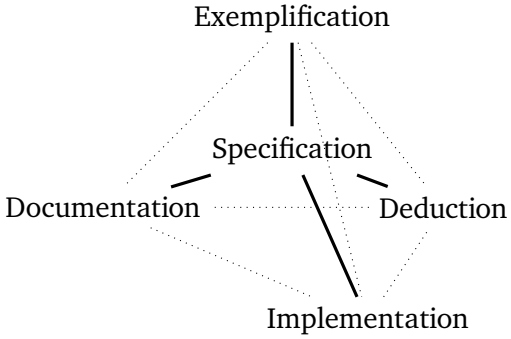

Figure 1 The Tetrapod Model

Aspect	Objects	Application
Specification	theories	ontology, mediation
Deduction	formal proofs	verification
Implementation	programs	execution
Exemplification	concrete data sets	testing
Documentation	informal texts	human understanding

Table 1 Aspect Overview

2 The Tetrapod Model via a Simple Example

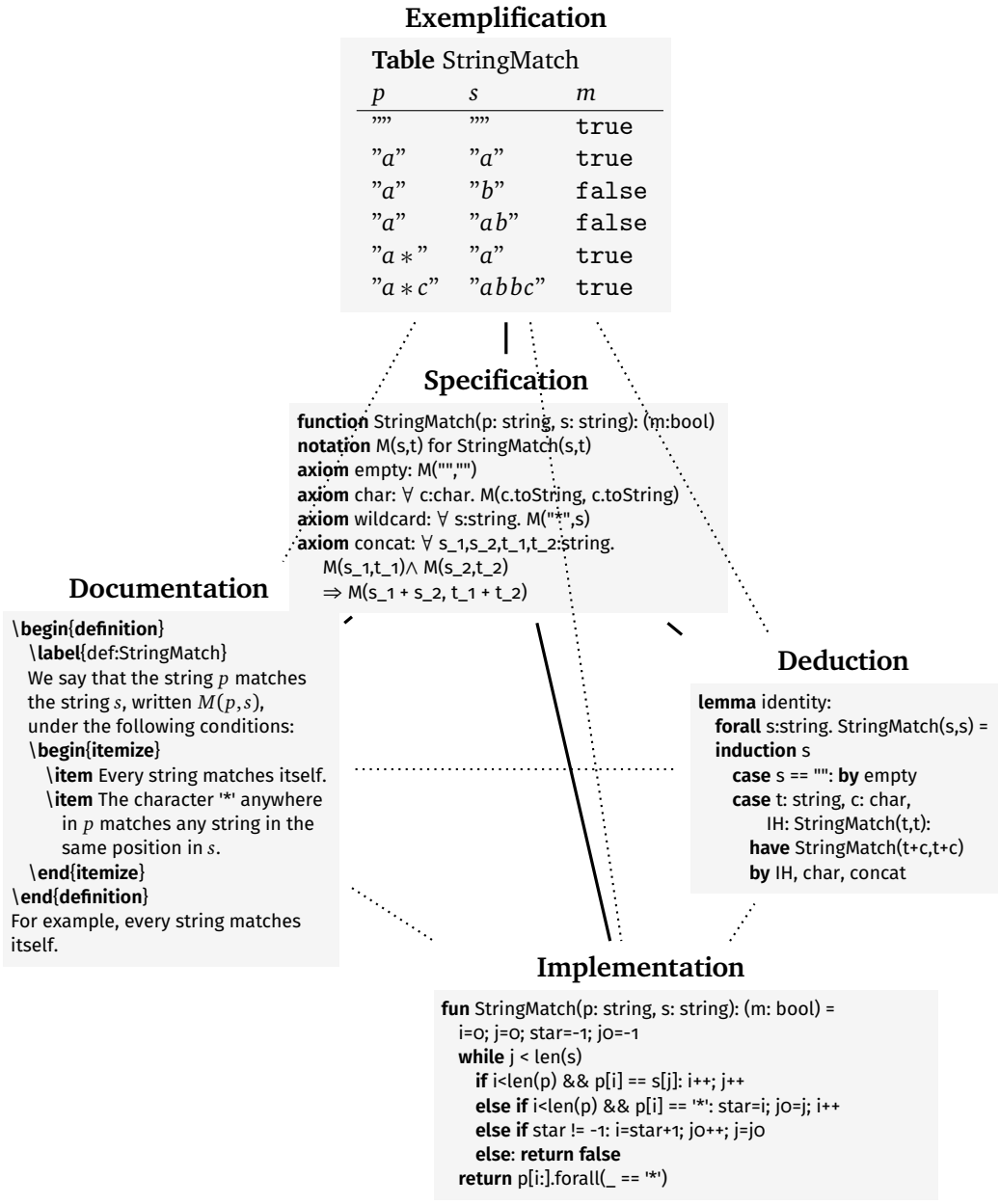

Figure 2 String Matching in the Tetrapod Model

We use a wildcard pattern matching function for strings as an example for the five aspects from Figure 1 and their interconnections. Figure 2 shows it in five parts, each corresponding to one aspect: an axiomatic specification, an informal documentation, a set of unit tests, an efficient implementation, and the proof of a high-level invariant that is implied by the specification and mentioned in the documentation. We assume five different pseudo-formats for neutrality and simplicity.

Figure 1 uses a tetrahedron structure to arrange the aspects with specification in the middle. It describes the input and output of the function and its intended semantics. Here we take a broad view on specification and include, e.g., logical languages like CASL [CoF04], object-oriented modeling languages like UML [UML17], and ontology languages like OWL [OWL09], all of which have in common that they formally describe some abstract properties of the types and functions without committing to a particular concrete realization.

We place specification in the center because it is special in that it provides the most concise and abstract description. In particular, it introduces names (of, e.g., types, functions, axioms) and signatures that are reused by the other aspects. The other aspects can be understood through how they add to, refine, and realize the specification.

The documentation part explains the semantics of the algorithm in a human-oriented way. In LaTeX this could be the text on the left in Figure 2. Note how the definition is tied to the name introduced in the specification via the label.

The implementation gives an efficient algorithm for the function. Depending on the programming language, this could look something like in lower part of Figure 2. Note how the type and function names tie the program to the specification.

The exemplification part provides a set of input-output tuples that can be used for illustrating corner cases, memoizing the function, or unit-testing an implementation. Using a relational database, this could be organized as a table as we see at the top in Figure 2. Note how the table and column names tie the data to the function, input, and output names introduced in the specification.

The deduction part applies logical proofs. This can be used to derive correct-by-construction algorithms, e.g., by verifying the implementation against the specification. But it can also prove additional properties that are implied by the specification and often used for high-level documentation. One example is shown on the right in Figure 2. Note how the function and axiom names of the specification are used in the proofs.

String matching may seem like a rather trivial example, but already case-*insensitive* matching of Unicode strings is an extremely complicated function (because Unicode case-folding is very complex, including e.g., locale-specific behavior or changes in string length) where leveraging all 5 aspects becomes important. In fact, we originally planned to use the case-insensitive version here, but it became too complex to work out as a short example.

3 Complementary Strengths and Weaknesses

The tetrahedron structure of our model gives us a convenient vocabulary for aspect integration: the four corner aspects can be seen as divergent ways for realizing and expanding on the specification. And the edges and faces correspond to combinations of two resp. three aspects. The corner aspects employ separately evolving and optimized methods that differ not only in the languages and tools but in the fundamental assumptions and goals. Indeed, we can find characteristic joint advantages for every combination of corner aspects.

Aspect	Role in efficiency	Role in correctness
Deduction	safe optimization	verification
Implementation	fast execution	differential testing
Exemplification	memoization	unit testing
Documentation	intended use	code review

Table 2 Aspects and their Roles

These advantages allow the corner aspects to contribute in different ways to achieving the key properties of efficiency and correctness. Table 2 gives an overview. For example, implementation's main advantage is to obtain fast execution. But it also contributes to correctness by employing redundant implementations that are tested differentially. Similarly, deduction's main use is to verify correctness with absolute certainty. But it also contributes to efficiency because verified code can be optimized much more aggressively: if a sufficiently fine-granular specification is kept fixed, and a verified implementation is changed and reverified, then it is guaranteed that optimization introduced no bugs. Systematic collecting tables of input/output examples can be used both to speed up software by memoizing and to check correctness by unit testing. Finally, documentation is the basis of human code review for correctness; and good documentation of how often and how the entry points of the code will be called is necessary to choose the most efficient data structures.

Note how the specification aspect is critically linked to each corner aspect: it provides the reference point for formal verification, an executable specification can be the reference point for differential testing, the specification-driven example generation can be used to obtain unit test suites, and a good specification can itself be or be part of a good documentation. Indeed, modern practice in safety-critical software development shows that all four corner aspects are powerful tools for obtaining correctness, and often the strongest method is to use multiple of them in parallel. Vice versa, bugs in the specification itself can be found by trying to formally verify its claims (e.g., that a method's post-condition actually suffices to derive the class invariant), by applying unit testing against the specification, by differentially testing an executable specification against an independent implementation, or by code-reviewing the specification against the documentation.

Similarly, all corner aspects contribute with their independent strengths to software maintenance. Good documentation makes the creation and maintenance of program code easier, especially when allowing for LLM-generated code. A good example suite, which can be available much earlier than the other aspects, allows identifying corner cases bugs in the other aspects early, like in test-driven development. It also enables regression testing when comparing multiple versions of the implementation. A redundant, not necessarily efficient, implementation allows for model-based testing. And the goal of proving high-level properties and invariants about the software, which are often stable across versions, can guide all other aspects.

Each advantage comes at a price. Table 3 shows the characteristic advantage of every corner aspect (i.e., an advantage of an individual aspect) and of its opposite

Aspect	characteristic advantage of	
	corner	opposing face
ded.	certainty	ease of use
impl.	execution	well-definedness
exempl.	tabulation	abstraction
doc.	flexibility	formal semantics

Table 3 Characteristic Strengths of Each Aspect vs. its Opposite Aspect Triple

Aspect pair	characteristic advantage
ded./impl. doc./exempl.	rich meta-theory simple languages
ded./doc. impl./exempl.	theorems and proofs normalization
ded./exempl. impl./doc.	decidable well-definedness Turing completeness

Table 4 Characteristic Strength of each Aspect Pair vs. its Opposite Aspect Pair

face (i.e., an advantage enjoyed by each of the other 3 aspects). For example, the first row means that correctness is a unique advantage of the deductive aspect; dually, ease of use is its characteristic weakness, i.e., an advantage shared by the three aspects of the opposing face.

Similarly, Table 4 shows the characteristic advantages shared by every pair of aspects, which correspond to the six edges of the tetrapod. For example, the first two lines mean that deduction and implementation languages share the advantage of having a rich meta-theory and the disadvantage of being more complex compared to documentation and exemplification languages. In the sequel, we discuss these 4+3 match-ups in more detail.

Deduction vs. Others Deduction languages enable unambiguous and mechanically verified certification of correctness. This is absent in the other three aspects: Documentation languages can express (informal) correctness proofs, but these are not mechanically verifiable.

And implementation and exemplification languages typically do not even allow stating correctness properties, let alone proving them. There are some partial exceptions here, e.g., assertions in programming languages or data consistency conditions in database languages do allow for some degree of correctness certification. But these are usually computable (e.g., they do not quantify over infinite domains) and are used to check decidable conditions. (We revisit the increasing integration of deduction and programming languages below.)

Conversely, while somewhat subjective, deductive languages arguably pose the biggest difficulties for beginners to learn and the biggest cost for experts to apply.

Writing proofs generally tends to be harder and more time-consuming than writing theories, programs, tests, or text, and this explains why deductive languages are used much less in mainstream applications and are usually restricted to more safety-critical applications. The ascent of LLMs will chip away at this cost, however, enabling the automated generation of formal specifications and correctness proofs.

Implementation vs. Others Programming languages enable efficiently executable descriptions of functions. This is not prevalent in the other three aspects. Documentation languages allow descriptions of algorithms, but these are not directly executable, in fact not even guaranteed to be algorithmically executable in any way, let alone efficiently so. Deduction languages allow giving algorithmic descriptions, e.g., through rewriting, logic programming, or by being expressive enough to incorporate simple programming languages with a termination calculus. But even at their best, they are far from efficient. And languages for concrete examples tend to focus on static concrete data rather than executing functions on them. Other aspects may employ computation in some way, e.g., tactics in deduction systems or query languages for collections of examples, but these are computations about the knowledge expressed in them, not a formalization of computational domain knowledge.

Conversely, implementations of functions lack well-definedness guarantees in the sense that every accepted program indeed defines a total function of the stated type. Instead, well-formed programs may abort or not terminate at run-time. Deduction and documentation languages allow proving the well-definedness of every object expressed in them. And languages for writing concrete examples are usually so simple that every object is trivially well-defined.

More precisely, it is impossible to have a language that (i) guarantees that every computable function can be expressed, (ii) has a decidable well-formedness checker, and (iii) assigns to every well-formed function as its semantics a well-defined mathematical function. Programming languages sacrifice property (iii) (by allowing for run-time errors). We might even use this **three-way incompatibility** result to differentiate implementation from specification and deduction languages: specification languages sacrifice property (ii) (if we consider a specification to be well-formed only if it is consistent), and deduction languages sacrifice property (i) (if we consider a function to be expressed only if its termination is proved).

Exemplification vs. Others Exemplification supports the development of large test suites, efficiently organized as input/output tables. This allows querying for certain test cases and using them in continuous integration. In its most general form, we can realize them as databases holding input-output tuples in one table per function. Because current tools do not enable a tight integration between implementation languages and databases (in particular, they usually cannot share the data structures), this is currently not commonly done in practice despite the potential benefits.

Deduction, implementation, and narration languages also allow for collecting examples in some way, typically as a set of toplevel declarations together with some properties they satisfy or not satisfy. But they usually do not make the heavily restricted language in which these example are written explicit. For example, typical

test frameworks simply annotate a declaration as a unit test; and documentations included examples as a normal part of the text. Somewhat of an exception are deduction languages that try to answer queries based on the specification, such as Prolog-style logic programming languages, or by generating values from the type signatures, such as Quickcheck-style test generation [CHoo]. But these methods do not allow for the systematic maintenance and querying of examples, let alone to do so efficiently for large test suites.

Documentation vs. Others Any formal knowledge representation language must solve a fundamental trade-off: should the language allow expressing anything the user wants to express, or should the user's freedom be constrained in a way that can be leveraged to enable practical algorithms (such as execution, proving, or testing). Documentation languages can be seen as an extreme point on this spectrum: they target processing by a human brain rather than an artificial machine. Thus, the interpretation of documentation languages has full access to human reasoning capabilities, background knowledge, sense of context, and intuitions and thus allows virtually unlimited flexibility. On the flip side, documentation is not guaranteed to be meaningful or even well-formed in any rigorous way. It can assume that the human reader can fix all the errors, presuppose the necessary knowledge, and make all the implicit modifications needed for the description to make sense.

Conversely, all other aspects employ a formal grammar that systematically allows only a certain carefully chosen set of inputs for which a semantics can be given. This allows rigorously defining the semantics of all inputs and building tools that can check if the input is well-formed and, if so, compute its semantics.

Deduction/Implementation vs. Exemplification/Documentation Deduction and implementation languages usually employ much more complex grammars — usually based on explicitly represented term structures. These result from carefully negotiated tradeoffs between user-demanded expressivity and designer-required simplicity. The latter is necessary to retain the ability to specify, document, and implement complex meta-operations like proving/prove-checking and compilation/execution. Consequently, they admit rich, mathematical meta-theories such as soundness/completeness theorems for deduction languages and complexity analysis for implementation languages.

Conversely, exemplification and documentation languages use much simpler syntactic structures. Languages focusing on concrete examples use simple data structures that can be leveraged for indexing and fast querying. And while the *semantics* of narrative language has a very complex structure as well, this structure is usually not made explicit in its syntax and the semantics construction is a very complex process that might be partial or non-compositional. So their meta-theories tend to be very simple or much less mathematical in spirit.

Deduction/Documentation vs. Implementation/Exemplification Deduction and documentation languages allow stating and proving theorems. They do so, of course, in fundamentally different ways: deduction uses formal languages with tool support for finding and checking proofs, and documentation states theorems and proofs informally

and checks them only through a social process. In both cases, proofs are typically the hardest part of the process, and considerable effort is invested to optimize the proof verification process. The theorem can relate aspects to each other, such as proving an implementation correct, but they can also go beyond, e.g., by establishing high-level invariants that are required by the documentation and implied by but not explicitly part of the specification or implementation, or by justifying program transformations and modular decompositions that enable clearer design.

Implementation and exemplification languages are significantly simpler by foregoing proofs entirely. Some programming languages allow stating theorems, e.g., as pre/post conditions or loop invariants, and some languages for large sets of concrete examples allow stating axioms such as via SQL consistency constraints or OWL assertions. But support is limited by practical needs and makes only first steps towards a multi-aspect system (cf. Section 4). Instead, these languages focus more on using expressions with computable values, i.e., ground formulas using only computable operators. That way they gain a normalization property: objects can be normalized simply by computing their value. This is usually trivial in languages for concrete data, and it is the main point in programming languages, where normalization amounts to running an expression. In particular, this enables comparing basic objects for equality, something that is undecidable in documentation and deduction languages.

Deduction/Exemplification vs. Implementation/Documentation Recall the three-way incompatibility discussed above: decidable well-definedness is the combination of properties (ii) and (iii), and Turing-completeness is property (i). Both exemplification and deduction languages achieve decidable well-definedness, albeit in very different ways: exemplification languages systematically use inexpressive languages in order to stay so simple that basically all data allowed by the grammar is well-defined or easily checkable for well-definedness. And deduction languages restrict expressivity far less and instead add a proof calculus that allows stating and proving that an object is well-defined.

Conversely, implementation and documentation languages sacrifice decidable well-definedness, also in different ways: the former needs extending with a deduction language to reason about well-definedness, and the latter does not allow for algorithmically checking its well-definedness proofs. But both gain the ability to express every computable function (Turing-completeness).

Note that well-definedness in deduction *tools* is often *practically* undecidable because the tool attempts to find the well-definedness proof automatically, which is undecidable. But we are assuming here that a definition consists of the pair of definiens and well-definedness proof, and checking the well-formedness of such a pair is decidable. Moreover, note that modern proof assistants are usually Turing-complete in the sense that they allow stating every computable function. But its calculus is not able to prove the well-definedness (in particular the termination) in every case, and even when it can, the tool cannot always find the proof automatically (cf. the halting problem).

4 Towards Multi-Aspect Software Systems

Ideally, we would build a system that combines the complementary strengths of all aspects. This idea is not new. In fact, many research efforts in software engineering can be understood as examples of developing multi-aspect systems.

Most commonly, programming language-centric software ecosystems tend to have connections to specification languages (e.g., code generation from UML) or integrate explicit abstraction mechanisms (e.g., C header files or Java classes), documentation frameworks (e.g., generation of API documentation in HTML), exemplification (e.g., unit test definitions alongside the main code base), and deduction languages (e.g., invariant annotations or code generation from proofs assistants). But the connections tend to be loose and ad hoc.

At the conceptual level, the V model [Roo86] can be seen as a systematic attempt to organize all five aspects, albeit with computation in the center and the other four aspects described based on how they relate to the program development workflow. The Tetrapod model differs in that it tries to provide an aspect-neutral map that can be used to classify and compare multi-aspect approaches, and to evaluate them from the perspective of a hypothetical ideal system that naturally integrates all aspects. We discuss some examples in the sequel.

There is a huge field of work on **combining computation and deduction**. Within that field, we can subcategorize systems by placing them along the computation-deduction edge, seen as a spectrum. Near the deduction corner, we can place systems strongly rooted in logic that try to support computation, such as

- many proof assistants, whose logics are expressive enough to subsume pure functional programming languages, often including a code generation facility such as Isabelle [Pau94] or Coq [Coq15],
- rewrite systems [DP01] that use directed equations as a way to represent computations,
- logic programming systems that represent computations as proof search such as Prolog or λ-Prolog [MN86].

Somewhat in the middle, we can place verified software development systems like Dafny [Lei10] or Why3 [Bob+11] that use an integration of logic and programming language. They usually generate proof obligations for completing the verification in dedicated deduction systems and mainstream programming language code for execution. Towards the computation corner, we can place deductive extensions of mainstream programming languages. These usually a specification of the semantics of a fragment of the programming language coupled with a verifier for them such as the KeY system for Java [Ahr+05], SPARK for Ada [Cha+24], or Verus for Rust [Lat+24].

All edges can be seen as spectra in this sense, albeit with some more populated than others. For example, there are lots of approaches that combine documentation with computation such as literate programming [Knu84] and API documentation generators. But there is much less systematic research on combining, e.g., deduction with exemplification such as logic-driven test-case generation as done in, e.g., [Dis+24]. Similarly, the **center-corner lines** are spectra. For example, model-driven software

engineering languages such as UML [UML17] can be placed along the line from specification to computation. And specification languages with a strong logical foundation like CASL [CoF04] can be placed on the line from specification to deduction. Database languages like SQL can be seen as combining specification (the database schemas) with exemplification (the database content and querying).

These examples are **by no means an exhaustive list**. In fact, the space of multi-aspect systems deserves an extensive survey article, which the Tetrapod can help structure but which would go far beyond the scope of this article. But critically, we observe that many multi-aspect tools are rooted in one aspect and then expand to accommodate others. Often this leads to tools whose design is biased towards their "home aspect".

Major tools that systematically try to integrate all aspects are rare. Arguably, software IDEs are inching closer, albeit in an implementation-centric way. Modern IDEs integrate, e.g., implementation with specification via code generation from UML, exemplification via unit testing frameworks, deduction via generating proof obligations for software verification tools, and documentation via automated documentation generation. This provides a strong integration at the user interface level, but the integration at the language and tool level still stays loose: typically different languages, each with their own IDE plugin, are used for each aspect, and it is not clear how to best organize the connection between them. Often structured comments (e.g., to annotate documentation, invariants, or unit tests) are used to link parts of the program to corresponding developments in other aspects.

5 Outlook

Despite many efforts and the enormous potential benefits, there is currently no single language or system in which all 5 aspects of our motivating example could be spelled out equally elegantly and in an interconnected way (which is why the alignment of identifiers like `StringMatch` across aspects is important).

It remains open

- what a truly tetrapodal system would look like, if it can be built, and if it would be successful if it were built,
- or if a loose integration of different languages, maybe in a common IDE, is preferable, and if so, how such a loose integration can be mediated best.

There is value in keeping the aspects separate: all aspects have individual advantages, leveraging which may require fundamentally different and thus mutually incompatible optimizations of the languages and tools; it is also valuable to allow development on all aspects to proceed in parallel without blocking each other, e.g., with different teams developing specification, implementation, verification, tests, and documentation in different repositories and different timelines. But often aspects are kept separate or integrated loosely simply because there is no other option: cross-aspect integration is often only an afterthought, and existing languages and tools cannot easily be

retrofitted with other aspects. This is especially true in the common case where cross-aspect integration becomes an issue only at a time when projects have already reached large scales of single-aspect development.

But we conjecture that in the long run, language and tool design will increasingly target the aspect integration goal, and our model helps make the benefits and challenges precise.

References

[Ahr+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. "The KeY Tool". In: *Software and System Modeling* 4 (2005), pages 32–54.

[AWS25] 32 Authors at Amazon Web Services. "Verified Authorization at Cloud Scale". In: *International Conference on Software Engineering*. to appear. 2025.

[Bob+11] F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pages 53–64.

[Car+21] Jacques Carette, William M. Farmer, Michael Kohlhase, and Florian Rabe. "Big Math and the One-Brain Barrier – The Tetrapod Model of Mathematical Knowledge". In: *Mathematical Intelligencer* 43.1 (2021). DOI: 10.1007/s00283-020-10006-0.

[CHoo] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *International Conference on Functional Programming*. Edited by M. Odersky and P. Wadler. 2000, pages 268–279.

[Cha+24] R. Chapman, C. Dross, S. Matthews, and Y. Moy. "Co-Developing Programs and Their Proof of Correctness". In: *Communications of the ACM* 67.3 (2024), pages 84–94.

[CoF04] CoFI (The Common Framework Initiative). *CASL Reference Manual*. Volume 2960. LNCS. Springer, 2004.

[Coq15] Coq Development Team. *The Coq Proof Assistant: Reference Manual*. Technical report. INRIA, 2015.

[Dis+24] C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, J. Kastner, A. Mamat, M. McCutchen, N. Rungta, B. Shah, E. Torlak, and A. Wells. "How We Built Cedar: A Verification-Guided Approach". In: *Foundations of Software Engineering*. Edited by M. d'Amorim. 2024, pages 351–357.

[DP01] N. Dershowitz and D. Plaisted. "Rewriting". In: *Handbook of Automated Reasoning*. Edited by J. Robinson and A. Voronkov. Elsevier and MIT Press, 2001, pages 535–610.

[Gin+11] D. Ginev, H. Stamerjohanns, B. Miller, and M. Kohlhase. "The LaTeXML Daemon: Editable Math on the Collaborative Web". In: *Intelligent Computer Mathematics*. Edited by J. Davenport, W. Farmer, J. Urban, and F. Rabe. Springer, 2011, pages 292–294.

[Knu84] D. Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (1984), pages 97–111.

[Lat+24] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. Lorch, O. Padon, and B. Parno. "Verus: A Practical Foundation for Systems Verification". In: *Symposium on Operating Systems*. Edited by E. Witchel, C. Rossbach, A. Arpaci-Dusseau, and K. Keeton. 2024, pages 438–454. URL: https://doi.org/10.1145/3694715.3695952.

[Lei10] R. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Edited by E. Clarke and A. Voronkov. Springer, 2010, pages 348–370.

[MN86] D. Miller and G. Nadathur. "Higher-order logic programming". In: *Proceedings of the Third International Conference on Logic Programming*. Edited by E. Shapiro. Springer, 1986, pages 448–462.

[OWL09] OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation. World Wide Web Consortium (W3C), Oct. 27, 2009. URL: http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. Volume 828. Lecture Notes in Computer Science. Springer, 1994.

[Roo86] P. Rook. "Controlling software projects". In: *Software Engineering Journal* 1.1 (1986).

[UML17] Editors of UML. *Unified Modeling Language*. Technical report. Object Management Group, 2017. URL: https://www.omg.org/spec/UML.

About the authors

Michael Kohlhase michael.kohlhase@fau.de
https://orcid.org/0000-0002-9859-6337

Florian Rabe florian.rabe@fau.de
https://orcid.org/0000-0003-3040-3655