Taming the IT systems complexity hydra

Olivier Zendra, Koen de Bosschere

The complexity of IT systems is a tremendous, costly and growing issue. By steering the way we develop these systems towards appropriate tools and methodologies, we'll be able to tame this IT complexity hydra.

Taming the IT systems complexity hydra

By OLIVIER ZENDRA and KOEN DE BOSSCHERE

Although most people remain unaware of its presence in the background, the ever-increasing complexity of IT, with its multiple sources, has been an ongoing issue for quite some time. It can even be qualified as a crisis, in both hardware and software. Indeed, this complexity has reached the point where systems are no longer fully understandable by human beings, which raises the question of how we can continue being in full control of their functioning. It is of course a matter of cost for the IT industry. But a number of incidents caused by bugs or a misunderstanding of some part of an IT system have already occurred. With an IT world that is permanently connected on a worldwide scale, the risk of damage caused by the lack of control of IT systems is both real and growing, with errors and malevolent attacks the most likely culprits.

Taming the IT complexity hydra is thus more necessary than ever. Fortunately, various solutions can be proposed to tackle the various heads of the hydra (i.e. the various aspects of complexity); these are solutions based on existing methodologies, tools and resources or extensions thereof.

Key insights

- IT systems complexity is high and ever increasing.
- IT complexity is threatening the quality and control of crucial systems that can affect the lives of many businesses and people in the EU.
- Taming IT complexity is vital for quality (safety, security, performance, sustainability, trustability, resilience) and cost (time to market and maintenance), hence competitiveness of EU industry.
- There is no silver bullet against complexity, not even AI.
- Modularity is key to mastering complexity. Modularity demands components, containers, contracts, specifications, services and orchestration.
- Formal methods, models, can harness (part of) IT systems complexity.
- The EU, like the rest of the world, is in dire need of educated, highly skilled IT specialists.

Key recommendations

- The EU should support efforts to tame IT complexity, for the sake of quality (safety, security, performance, sustainability, trustability, resilience) and cost (time to market and maintenance), hence competitiveness of EU IT industry.
- The EU should promote research on methods and tools on modularity, components, containers, contracts, specifications, services and orchestration.
- The EU should promote research on formal methods and tools for modelling IT systems and their functional properties (what they do, the algorithms) as well as their non-functional properties (how they do it: time, energy, security...).
- The EU should train more highly skilled IT specialists.

Image: ID 124451653 ©Pop Nukoonrat Dreamstime.com

The complexity of IT systems, on both the hardware and the software side, keeps growing exponentially and creates an ever-bigger challenge. It is already the case that some systems can be considered as *no longer completely understandable*, hence no longer mastered, not only by their users but above all by their designers, developers and maintainers. This state of affairs cannot go on, so *it is crucial for the EU that complexity be mastered*, for users, by its IT system providers, in all its dimensions.

This article provides an overview of the many sources of complexity that make it similar to the mythological hydra, and presents solutions we deem important to cut off its ugly heads and/or tame this beast.

IT system users don't like complexity: developers must hide it

From the user point of view, *complexity* has to be hidden so as to provide an easy, pleasant user experience. IT systems have very much improved and even done well to hide the nitty gritty details for basic levels of use but the complexity for users tends to move to higher levels of use. Users increasingly want to have access to multiple functions and services, from various providers, spread all across the world, and all of this at the same time, possibly on multiple and varied terminals (from smart watches to desktops, via smartphones and tablets), presented to them in a simple and convenient way.

Multiple installed applications go against simplicity. Users need *as-a-service meta-applications*, that is to say, aggregators of multiple applications, to save them from the connection and coordination issues associated with accessing the various applications. In addition, these aggregators cannot just present users with a juxtaposed view of the various application results: they must be smart integrators that process and manage the complexity of the various results and present them in a more synthetic, and easier to understand way. Good examples of such meta-applications, which currently tend to be domain-specific, are price aggregators and comparators (for travel, hotels, etc.), and virtual personal assistant capabilities like Alexa Skills.

IT systems are full of complexity: the sources are varied

For IT system developers, *complexity springs up in all corners of IT systems development*, for both **hardware** and **software**, and its *various aspects call for different kinds of solutions*. This section presents an overview of the root sources of IT complexity that we deem important.

In **hardware**, since Dennard scaling has stopped, processor systems have become tightly-interconnected multi-cores (exposing *parallelism* with and without *concurrency*), fitting in an increasing number of accelerators (exposing *heterogeneity*), aggregated in variably deployable units (exposing *statelessness*) and networked (exposing geographical distribution and decentralization), for ease of access via the web (exposing *asynchrony*). Field-programmable gate arrays (FPGAs) are based on radically *different programming models*. Hybrid platforms are also emerging. *Heterogeneity* is thus probably more prevalent than was generally expected. As a consequence, the hardware environment is *evolving extremely rapidly*, even faster than in the era of Moore's law.

This increasing hardware complexity is now an emerging crisis. The (incomplete) documentation amounts to 9000 pages, with a table of contents of 100 pages, is written in informal English and periodically amended by errata. This, for many chips, every few weeks [3]. How can humans cope with such a *diluvial amount of information*? Is this huge engineering effort worth it? Since hardware has no formalized semantics, how can we ensure it is correct? Is verified software built on the shifting sands of possibly incorrect hardware? Bugs occur, safety or security breaches too, making attacks by various aggressors easier.

This explosion of complexity is matched in **software**, which contains many sources of complexity at all levels of the software stack.

The system development ecosystem is an immense maze consisting of scores of *methodologies and their derived tools (Figure 1)*.

Programming languages alone are also a source of complexity (Figure 2). Historically, statically typed languages used to be the most popular ones. Then dynamic languages became popular (Ruby leading). Now languages are more mixed [4]. As a result, today *more than 8000 languages exist*, from the generalist languages addressing a wide range of needs to more specialized, targeted languages, or even DSLs (domain specific languages) fully tailored to one specific domain of application. A large number of them are still in active, live use, creating a modern-era technological Tower of Babel:

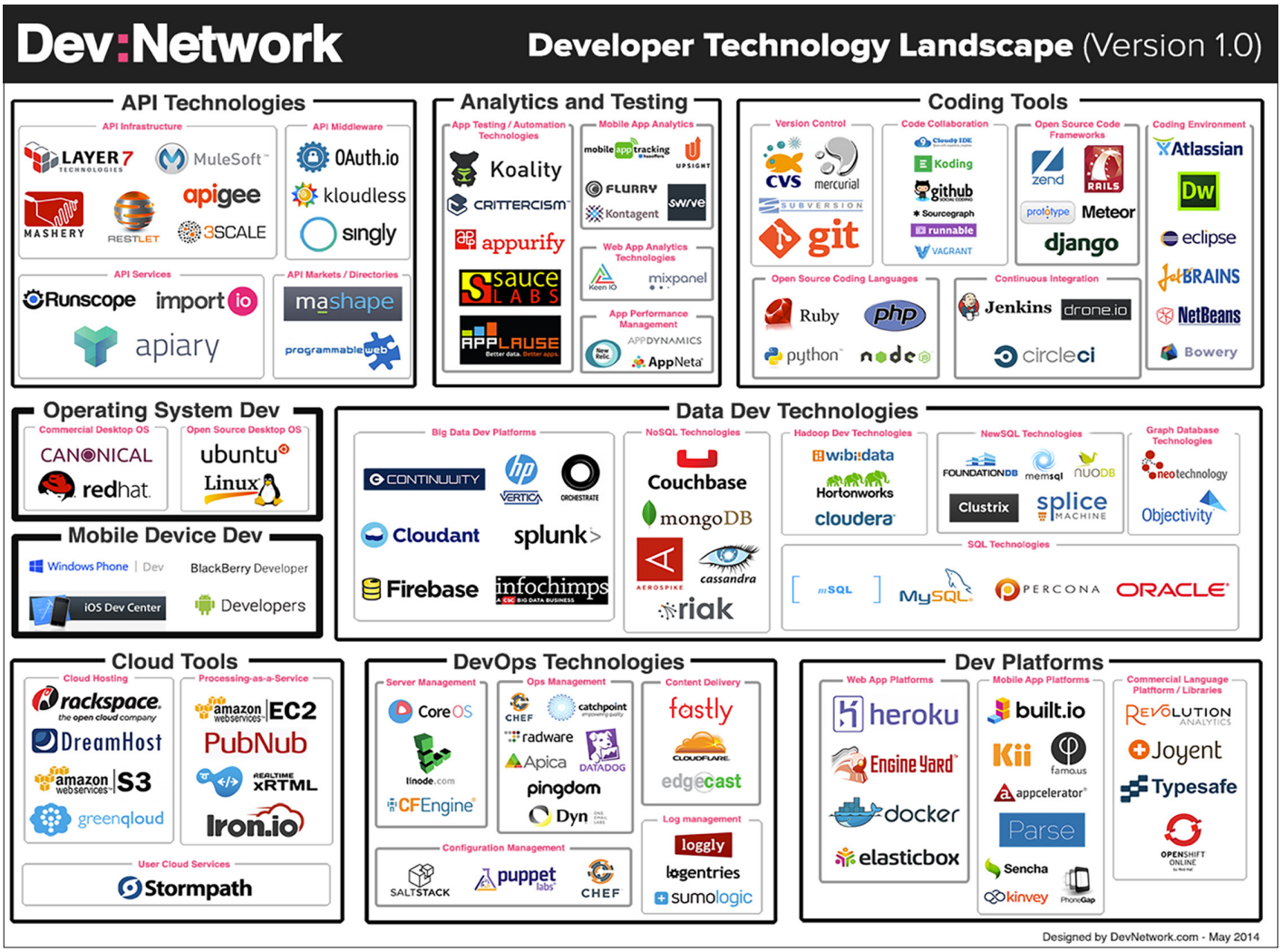

Figure 1: A sample of the developer's technology landscape in 2014. Things have not improved since. Source: DevNetworks Sought enhancements: asserting correctness.

This *multiplicity and heterogeneity of languages* also creates huge complexity, in terms of their interaction and developers' ability to master them, etc. What are the tools available to manage this complexity when using different languages in the same codebase (e.g. how do you diff)?

At the same time, the various languages exist for a reason, while addressing different needs. A good illustration here are DSLs that help tackle the peculiarities of some application domains in better ways than general purpose languages, hence in a simpler way.

Software libraries provide off-the-shelf capabilities, or features, that developers like to reuse to build systems from existing parts rather than reinventing the wheel. Yet the library ecosystem is huge. Which library to choose when wanting to add a feature is a rather informal, ad-hoc process, and may be a complex one when faced with many alternatives. Library versions also have to be taken into account, because compatibility issues between versions of the various libraries create an intricate web of dependencies.

Application code itself can be quite complex, just because of the inherent complexity of the problems it solves. Admittedly, this kind of complexity seems more useful than certain other kinds, yet it has to be managed. Application code complexity can also come from the coding style and/or the language used. Terseness can sometimes lead to obfuscation, while some verbosity may help understanding, hence lower complexity (see Figure 3). A balance thus has to be found.

In addition, with the growing pervasiveness of the use of computer systems in virtually every aspect of our daily life, the production side of the IT community is faced with complexity factors that add to the classical functional complexity. These factors comprise *non-functional properties* such as energy, time and other resource constraints, ever-advanced human-computer interaction, the weaving of cyberspace into physical reality, as well as continuous delivery within continuous operation. All this results in many

Language Ranking: IEEE Spectrum

Rank	Language	Score
1	Python	100.0
2	Java	95.3
3	C	94.6
4	C++	87.0
5	JavaScript	79.5
6	R	78.6
7	Arduino	73.2
8	Go	73.1
9	Swift	70.5
10	Matlab	68.4
11	Ruby	66.8
12	Dart	65.6
13	SQL	64.6
14	PHP	63.8
15	Assembly	63.7
16	Scala	63.5
17	HTML	61.4
18	Kotlin	57.8
19	Julia	56.0
20	Rust	55.6
21	Shell	52.0
22	Processing	49.2
23	C#	48.1
24	SAS	45.2
26	Cuda	41.0
27	Visual Basic	40.3
28	Objective-C	38.9
29	Delphi	38.6
30	Perl	38.2
31	Verilog	37.6
32	VHDL	36.7
33	LabView	36.7
34	Elixir	35.8
35	F#	34.7
36	Prolog	34.6
37	Lua	34.4
38	Lisp	33.0
39	Ada	32.8
40	Apache Groovy	32.0
41	Scheme	31.4
42	Haskell	30.8
43	Cobol	30.4
44	Clojure	29.8
45	ABAP	29.5
46	D	27.7
47	Forth	23.7
48	Ocaml	23.7
49	TCL	22.1
50	LadderLogic	19.5

Figure 2: IEEE Spectrum Top 50 programming languages 2020 [8]. Python's success may lie in its ease of use to "glue" together libraries, where most of the computation are done, and its large number of supporting libraries, covering many domains.

```
define F (getchar()&15)
#define v main(0,0,0,0,
#define Z while(
#define P return y=~y,
#define _ ;if(
char*l="dbcefcbddabcddcba~WAB+ +BAW~          +-48HLSU?A6J57IKJT576,";B,y,
b,l[149];main(w,c,h,e,S,s){int t,o,L,E,d,O=*l,N=-1e9,p,*m=l,q,r,x=10 _*l){y=~y;
Z--O>20){o=l[p=O]_ q=o^y,q>0){q+=(q<2)*y,t=q["51#/+++"],E=q["95+3/33"];do{r=l[p
+=t[l]-64]_!w|p==w&&q>1|t+2<E|!r){d=abs(O-p)_!r&(q>1|d%x<1)|(r^y)<-1){_(r^y)<-6
)P 1e5-443*h;O[l]=0,p[l]=q<2&(89<p|30>p)?5^y:o;L=(q>1?6-q?l[p/x-1]-l[O/x-1]-q+2
:0:(p[l]-o?846:d/8))+l[r+15]*9-288+l[p%x]-h-l[O%x];L-=s>h||s==h&L>49&1<s?main(s
>h?0:p,L,h+1,e,N,s):0 _!(B-O|h|p-b|S|L<-1e4))return 0;O[l]=o,p[l]=r _ S|h&&(L>N
||!h&L==N&&1&rand()){N=L _!h&&s)B=O,b=p _ h&&c-L<S)P N;}}}t+=q<2&t+3>E&((y?O<
80:39<O)||r);}Z!r&q>2&q<6||(p=O,++t<E));}}P N+1e9?N:0;}Z l[B]=-(21>B|98<B|2>(B+
1)%x),++B<120);Z++m<9+l)30[m]=1,90[m]=~(20[m]=*l++&7),80[m]=-2;Z p=19){Z++p<O)
putchar(p%x-9?"KQRBNP .pnbrqk"[7+p[l]]:x)_ x-(B=F)){B+=O-F*x;b=F;b+=O-F*x;Z x-F
);}else v 1,3+w);v 0,1);}}
```

Figure 3: Complexity is not only depending of the size of the code, this example shows a complete program. How it works was explained in a book of 170 pages. Can you guess what it is doing? [9]

systems being composed of complex webs of dependencies that are easy to break and hard to maintain. Furthermore, there are still issues that have not been completely solved by the IT community. Among them, *how do we measure and value software quality?* Which non-functional properties or metrics have to be considered? *How do we value non-functional properties like speed, low-energy, high-security?*

Another of the crucial and complex aspects in the IT ecosystem is the importance and variety of legacy. Legacy is the heritage of the past, composed of *existing operating systems, libraries, languages, development tools and hardware*. Legacy represents a huge amount of code, estimated in 2000 at over 100 billion lines of code, most of it COBOL [6,7]. Legacy hinders the taking of new directions, yet it cannot just be done away with. Indeed, the service provided by large existing legacy systems must still be provided, so disrupting them is not an option. New languages and libraries must be able to interoperate with legacy ones. New software must often run on old hardware or old OSes, which multiplies the possibilities and tests to be done, the compatibility patches to write, for no other usage than having an IT system work in yet another particular context.

Furthermore, in order to reduce cost and time to market, it is much better and very common to reuse existing elements – even code parts found in public repositories on the web – to extend or modify existing systems, than to start each development from scratch. Indeed, legacy code, despite all its drawbacks, still makes it possible to tackle problems without reinventing the wheel, by reusing old, tested and tried, libraries that have been very fine-tuned and well debugged over the years, thus removing a lot of the complexity of new developments.

The issue there is thus not so much the existence of legacy as its intrinsic *quality* and the complexity to *integrate* it into new developments. Tradeoffs are thus of the essence when reusing legacy code.

Figure 4: Caeretan hydria, c.525 BC, Hercules slaying the Lernean hydra, Collection of the J. Paul Getty Museum, Malibu, California. Image: Wolfgang Sauber: Getty Villa, CC BY-SA 3.0, creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons

Overall, all these sources of complexity add up, as do the size of the elements composing the system: *the bigger the system in terms of functionalities, the greater its complexity*. Similarly, outside of the purely technical complexity, the greater a system, the greater the team needed to develop it, the greater the complexity of the development process and its management.

Impacts of complexity

The consequences of complexity in IT systems, coming from the above-mentioned sources, are very *simple: high levels of complexity mean high costs, and high risks.* High complexity brings high development costs, because of the size of the teams needed to develop the systems, and of the time needed to do so. It also incurs high risk of delays in the process, risk of poor quality in the system (risk of bugs leading to malfunctions), lack of speed, lack of safety and lack of security. The same applies of course for ongoing maintenance of complex existing IT systems, and for their evolution, with added difficulty that the original knowledge of the system designers and implementers is often gone.

In a nutshell, complexity must be tamed, in order to keep the IT system under control.

Fortunately, various solutions exist, or are within reach with reasonable efforts, to tackle the various heads of the complexity hydra.

Modularity does manage complexity, with additional benefits

Modularity at various levels is the main key found by humans to mastering complexity and to the reuse and integration of hardware or software legacy, especially across programming languages. The elements to (re)use have to be properly architected so as to be taken as whole *modules*, properly *contained* and *encapsulated*. On the software side this implies isolating the implementation inside the proper module (class, object, library, container…) and exposing only the right amount of interfaces at the boundaries of the module, to provide (micro-)services. Clear *contracts* must thus explicitly define the behaviour of the *interfaces* exposed at the boundaries, both inward and outward. Enhancing module/container interface *specifications*, so that they help assess semantic conformance at build, integra-

tion, deployment and execution time is necessary to properly achieve these goals. It should take the form of enforceable contracts covering both *functional* (i.e. the algorithms) and *non-functional* (e.g. time, power and energy, security and safety, etc.) properties.

This kind of modularity would be especially apt to current IT systems, which are generally extremely connected and distributed over the web. It matches very well with the *microservice* paradigm, an enabler of modern, heterogeneous software composition. Indeed, an individual microservice is a small self-contained application that has a single responsibility (which gives it a clear and distinct role in a composition), a fully-self-contained and preferably lightweight stack (which allows its software dependencies to be always fully satisfied), and which can be deployed, scaled and tested independently (which facilitates software evolution) [1]. The "microservices" architectural style yields a single application from the coordination of a suite of unitary services [2], each of which exposes an application programming interface (API) *outside* of their codebase, which is invoked using *asynchronous* (crucial to loose coupling) *web-based* service requests (key to reachability). Microservices can run isolated from others in containers, using hypervisors to segregate them. This provides more resilience in case of hacking, since contaminations should be blocked between containers.

Software applications and infrastructures will increasingly be aggregates of heterogeneous artefacts with a variety of deployment requirements. Controlling them can hardly be done in a merely declarative way or scattered in a maze of uncorrelated and independent scripts. Languages and tools for *orchestrating* collaborative distributed and decentralized components are thus needed.

In addition to helping integrate different (possibly legacy) elements, modularity is key to boosting the repairability of IT systems. Being able to replace a part (be it a software or a hardware one) with another when it is found to be faulty, or when it becomes obsolete, is a powerful way to extend the lifespan of an IT system. Although this seems obvious, when thinking of e.g. automobile parts, this is a concept less developed for hardware in IT systems. Software parts are more often upgraded, with many OSes, libraries and applications having new versions released with patches and/or improvements, thanks to these updates being mostly automated, hence very easy, on the user side.

By repairing hardware parts or modules, IT system lifespan can be increased, thus decreasing their global ecological footprint both in terms of raw resources and carbon impact.

By patching/upgrading software, IT system quality can continuously be improved, thus avoiding the costs and inconvenience of faulty behaviour, especially with respect to safety and/or security.

Modularity is also key to more easily developing new IT systems, allowing reuse of hardware or software modules and making it possible to create whole new product lines with limited effort. A well-known example is printer product lines, which clearly rely on modularity and componentization to produce a large variety of similar but not identical products to tackle a variety of consumer needs, thanks to a limited set of common subparts. There, modularity clearly decreases financial expenditure and time to market, which provides several competitive advantages.

Like for reusability of (legacy) elements, modularity for repairability requires clear interface contracts, specifications, at module boundaries, since the mechanisms are the same. Again, these contracts and specifications must take into account the non-functional properties as well, so as to carry enough information to ensure the proper composability of the modules, especially in the long term, with various evolutions of the system, hence evolutions of the other surrounding modules. These contracts must also be easy for developers to master and to deal with, especially when taking into account the shortage of skilled IT professionals in the EU. At the same time, these contracts must be amenable to formal verification and/or proofs.

Carrying enough information, while at the same time providing a good *level of abstraction* to hide away the details and not go in the way of composition, is an issue that must be tackled. It implies being able to have different levels of abstraction in the models and the tools, so as to be able to zoom in or out, depending on the level of details needed at the level of composition considered. These levels allow different views, with more or less information being provided, while keeping the underlying information complete, with no loss.

Abstracting away complexity with formalization, models and tools

In order to provide these *different levels of abstraction*, and the expression of contracts at module boundaries, appropriate models of the systems have to be relied upon to cope with complexity.

There is a dire need for *formalized semantics* to facilitate better analysis of the system and its properties, and the derivation of formal proofs for (at least some of) them. It is currently, however, near impossible, or at least prohibitively expensive, to mathematically formalize and completely prove large IT systems: they are too complex. However, we do know how to reason about functional correctness of programs and some smaller parts, modules, can be formalized and proven. This partial *verification* is currently the norm, to provide levels of assurance, mastering part of the complexity.

A lot of hardware has no formalized semantics. The hardware models for software development are thus inaccurately specified. So, it is difficult to formally ensure correctness: verified software would even in a way be built on (shifting) sands… Fortunately, this is changing. ISAs (instruction set architectures) are being (more) formally specified [3], and include ARM [10,11], RISC-V [12,13].

Software systems too will increasingly rely on formal methods. This is already happening; for example, the Isabelle proof assistant [14] is commonly used in the writing of seL4 OS [15], while Coq [21] is for the formal verification of the Compcert compiler [22]. Executable specifica-

Credit: ID 92496894 ©Nattapol Thepyasuwan | Dreamstime.com

tions put in the code (i.e. contracts) should also be increased, proving both a means to document the intent of the code, its specification, and to help its verification by automated tools. In addition, these specifications should provide information not only about the program functional aspects (what it does, the algorithms and functions), but also about what is currently called its *non-functional properties* (how it does it), like time and reactivity, power and energy, safety, security, etc. There lies a real current challenge: programmers and support tools should be able to express, manipulate, and reason about these non-functional properties, to yield static proofs of functional as well as non-functional correctness, to make runtime decisions, to support runtime assertions to check that the necessary properties hold during execution, and that they have adequate semantics to handle violations so that safety conditions are restored.

Efforts along these lines already exist and must be supported. It is necessary at the same time to also pursue less formal but more practical efforts aiming to improve the quality of the developed system elements and modules, in a very concrete and practical way, helping developers cope with some parts of the complexity. The MDE (model driven engineering) methodology [16], including the well-known UML (unified modeling language) [17] adopted by the OMG (object management group) computer industry standards consortium [18], and the related modeling tools (e.g. the Eclipse Papyrus Modeling Environment [19]), have been for a long time making progress in that direction and should be supported.

However, the advance of formal methods in IT systems has been hindered by past and current market realities. Indeed, business constraints (time to market, cost of production) and the programmers' mindset have generally focused on delivering functionalities to customers, since this is what sells. Integral correctness is rarely pursued by design; more often it is sought as a product of quality assurance activities, either performed retrospectively or in parallel to development, but not sufficiently ingrained in it. While some enterprises do specialize in providing tools that help the quest for correctness, their success has never even remotely approached that of organizations providing functionalities to the end user, such as the likes of Facebook or Twitter. Still, the potentially negative impact of this situation is huge, for loss of value, increase of risk, and spread of threats, and should be acted upon with a more vigorous quest for quality. The fact that some very famous IT companies provide end-user licence agreements that, in essence, remove any responsibility on their part should the product not work, means that the cost of such failures falls to the customer rather than to the provider, which is a very uncommon practice in other business domains. *Regulations against this could strongly help the quest for quality, by putting a higher price on the damage caused by poor quality IT systems.* Mandating liability for IT systems should thus be a priority for the EU to boost the quality of its IT systems.

Coping with complexity of formal methods may be an issue for developers/designers, but it will be a simpler one to solve than directly managing the full complexity of hardware and software. With tools getting easier to use, good programmers should have no problem mastering formal tools.

To tame the hydra, you need tamers: the role of IT education

Mastering all this complexity indeed requires *many highly skilled IT specialists*.

Educated designers use proper design methods and tools, producing high qual-

ity architecture and modular systems. Educated programmers program well and produce good implementations, even with poor languages. It is a fallacy to believe that to build significant IT systems, uneducated programmers can simply use tools in a kind of copy-paste way, not fully understanding what they are doing and what are the fundamental underlying concepts, and yet still produce good quality systems. Learning and mastering the fundamental concepts is key to good decision making in IT system production.

"Controlling complexity is the essence of computer programming." [5]
Brian Kernighan

At the same time, it is necessary to have tools that can present the proper level of abstraction, hiding the details when they are not needed. Easy programmability is thus a must-have goal. Tools must improve. Graphical programming was an interesting track to ease programming and bring it to the masses to some extent, with visual programming languages [20]. However, so far, graphical programming is still not scalable, strongly limiting its usefulness in commercial application building. The UML modelling language, with its graphical representation, goes to some extent in the graphical programming direction (but certainly not for the masses), and has made it possible for professional IT systems developers and designers to better represent the systems, hence to master complexity in a better, yet still incomplete and imperfect, way.

Machine learning can also be seen as an interesting track to help in IT system production. It has the capacity to learn heuristics, hard-coded control loops, policies, and help implement them in an automated or semi-automated way, thus saving significant amounts of time. It could also help with some architectural choices, based on the specifications. However, we should not think machine learning will write all of our programs for us anytime soon. All the hard, system-wide problems will remain: security, correctness, reliability, availability. In addition, although AI or its currently fashionable incarnation, deep learning, could help take away some of the complexity of programming IT systems (chopping off one head of the complexity hydra) by for example writing automatically some parts of programs (e.g. [23]), the use of machine learning could make it much more difficult to analyze the correctness, hence the safety and security, of IT systems, thus increasing the complexity of these stages (thus growing news heads for the complexity hydra)... So all these problems will have to continue to be addressed mainly by skilled human IT specialists.

Unfortunately, the latter are in scarce supply in the EU, with numbers being insufficient to fulfil the needs of our economy; this greatly hinders EU innovation and competitiveness. The need for a sizeable community of educated professionals capable of developing IT systems and who understand the fundamental concepts that underpin IT systems, must be addressed.

Conclusion

The complexity of IT systems is a monstrous hydra that cannot be left unattended. To tame it, and therefore ensure the good quality of the EU's IT systems, as well as the competitive advantage of its IT systems providers, there is no silver bullet. On the contrary: the answer is multifaceted, as much as complexity is multifaceted. Concretely, the EU must steer the way we design, develop and maintain these systems in more modular ways, using the practical power of containers, encapsulation, contracts, microservices and orchestration, as well as the formal power of verification, proofs, and correctness checking methodologies and tools, and explore how new technologies such as artificial intelligence can help. The EU must also have a large workforce of people skilled in IT systems, to tackle the challenges of today and those of tomorrow.

References

[1] J. Thönes, Microservices, IEEE Software, 32(1):116, Jan 2015.
[2] Martin Fowler, Microservices, https://martinfowler.com/articles/microservices.html, Mar 2014
[3] Timothy Roscoe, HiPEAC Vision Consultation meeting, 7 April 2020.
[4] Tiobe language index. https://www.tiobe.com/tiobe-index/
[5] Brian W. Kernighan, P. J. Plauger. Software Tools. Addison-Wesley Publishing Company, 1976
[6] F. P. Goyla, Legacy integration-changing perspectives [Cobol], in IEEE Software, vol. 17, no. 2, pp. 37-41, March-April 2000, doi: 10.1109/52.841604.
[7] COBOL legacy, Wikipedia. https://en.wikipedia.org/wiki/COBOL#Legacy
[8] "The Top Programming Languages", https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020
[9] Toledo Nanochess C program: https://nanochess.org/chess3.html
[10] ARM Architecture: https://en.wikipedia.org/wiki/ARM_architecture
[11] ARM ISA family: https://developer.arm.com/architectures/instruction-sets
[12] RISC-V: https://en.wikipedia.org/wiki/RISC-V
[13] RISC-V specifications: https://riscv.org/technical/specifications/
[14] Isabelle proof assistant: http://isabelle.in.tum.de/
[15] Proofs in seL4: https://sel4.systems/Info/FAQ/proof.pml and https://docs.sel4.systems/projects/l4v/
[16] Model-Driven Engineering: https://en.wikipedia.org/wiki/Model-driven_engineering
[17] Unified Modeling Language: https://en.wikipedia.org/wiki/Unified_Modeling_Language
[18] Object Management Group: https://en.wikipedia.org/wiki/Object_Management_Group and https://www.omg.org/
[19] Eclipse Papyrus Modeling Environment: https://www.eclipse.org/papyrus/
[20] Visual programming languages: https://en.wikipedia.org/wiki/Visual_programming_language
[21] Coq proof assistant: https://en.wikipedia.org/wiki/Coq
[22] CompCert compiler: http://compcert.inria.fr
[23] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu and Neel Sundaresan, IntelliCode Compose: Code Generation Using Transformer. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020

Olivier Zendra is a Tenured Computer Science Researcher at Inria, Rennes, France.

Koen De Bosschere is Professor in the Electronics department of Ghent University, Ghent, Belgium.

This document is part of the HiPEAC Vision available at hipeac.net/vision.

This is release v.1, January 2021.

Cite as: O. Zendra and K. De Bosschere. Taming the IT systems complexity hydra. In M. Duranton et al., editors, HiPEAC Vision 2021, pages 100-107, Jan 2021.

DOI: 10.5281/zenodo.4719574

The HiPEAC project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 871174.