New Software Engineering Requirements in Clouds and Large-Scale Systems

Lutz Schubert, University of Ulm
Keith Jeffery, Keith G. Jeffery Consultants

Editor's Note: In this invited article, the moderators for the European Commission Cloud Computing expert working group discuss the changes necessary to the traditional von Neumann principles to support today's resource-intensive applications.

At the dawn of IT and the Internet, it could still be safely assumed that all of the main operations would take place on a single computing instance with all data colocated or at least more readily available than the operational power would actually require. Parallelism was already being used to deal with complex simulations in high-performance computing, but mainstream parallelism actually referred to resource sharing between multiple services on a local processing unit, that is, pseudo-parallel execution.

This environment gave rise to the traditional programming concepts by von Neumann, which build on the following main principles:

- Code is executed sequentially.
- Code isn't mobile.
- Data is colocated with code at no access cost.
- Code is executed as a single process- and datastream.
- Processes are executed in a single-tenancy model.

These principles are in contradiction to the modern cloud and Internet-based environment in which we live and the increasing issues in processor development, which is strongly based on distribution, sharing, parallelism, and so on. Modern software engineering principles tend to build up additional functionalities that try to compensate for the inherent problems of the traditional concepts, rather than actually addressing them.

As modern applications increasingly hit the network and memory walls, more and more software engineers turn to dataflow models on top of the traditional paradigms to artificially add information about data requirements, access speed, and so on, rather than rethink how data-oriented processing differs from John von Neumann's and Alan Turing's strictly computing-oriented view.

This article investigates the core challenges arising in modern and future IT, and their impact on traditional application programming and execution. We also look at current approaches to overcome the limitations of von Neumann in light of these challenges.

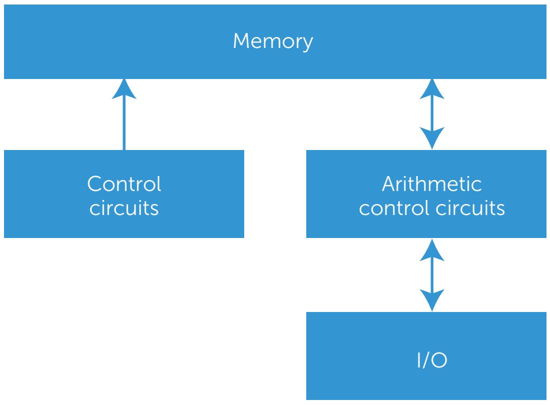


FIGURE 1. The processor architecture as foreseen by John von Neumann, which still guides the organization and behavior of modern processors.

The Von Neumann/Turing Concepts

Modern processors and software engineering methods are based on the computing concepts developed by Turing and the architecture to realize these concepts developed by von Neumann (see Figure 1). Without going into the full historical details, as well as the ensuing debates about the best approaches or later changes in the architecture details (Harvard, multicores, and so on), all of these systems share the following predominant factors:

- The actual work is encoded in a sequence of operations that are executed by a (central) processing unit. These operations are single instructions performing minimal actions on data requiring the last execution state as context.
- Although code and data are stored in memory and this storage unit can be explicitly addressed, the instructions act on immediate context—that is, the model doesn't require that the context is explicitly fetched first. Code and data are expected to be immediately available and can be manipulated.
- Anything outside this direct execution environment is considered to be an external device and therefore connected by a dedicated I/O. Notably, the requirements for larger versus faster storage have led to a flexible interpretation of this I/O, where the distinction between the internal bus and the external I/O becomes vague, leading to the well-known storage hierarchy model.

The storage hierarchy model can be classified along two primary factors: speed and means of access. We talk of I/O when the code needs to explicitly enact an access protocol, whereas we speak of memory and bus access when the logic is executed by the hardware and no mechanisms are needed by the code. This distinction is essential. Turing didn't foresee I/O being used as an essential part of the execution context, rather to represent devices the code can work with. The typical interaction with these devices is therefore either blocking (waiting for reply) or an asynchronous event (such as reacting to an event triggered by hardware; a keystroke, for example).

Challenges of Modern IT

Modern IT is predominated by four major factors that don't comply with the traditional paradigms.

Mobility

Computers aren't bulky stationary devices anymore, but have become a commodity that accompanies us throughout our daily lives. As a Cisco report shows, the number of connected devices is expected to continue to grow, with smart devices and connections steadily replacing nonsmart ones.[1] The traditional von Neumann principle assumes, however, that state, code, and data always will always be colocated, allowing no switching between devices.

Sharing

Applications, resources, and data are no longer isolated to a single machine but are constantly exchanged between users and devices, and are accessed and modified simultaneously. For example, social networking is currently the number one online activity of Americans, who spend more than half an hour each day on average on Facebook and other social networking sites, sharing data, pictures, and so on.[2] Von Neumann's principles don't account for data that's located remotely, let alone data that can be manipulated from different locations—potentially even with the same code.

```
X86 Assembly Instruction: Move
The mov instruction copies the src operand into the dest operand.
Intel Syntax:
        mov dest, src
Operands:
        src: Immediate, Register, Memory
        dest: Register, Memory
```

FIGURE 2. The x86 data access instruction.

End of Clock Speed

Modern processors aren't getting faster, but there are more of them—that is, more cores per processor, more processors per node, and so on. To continue to increase performance, parallelism is therefore essential, as opposed to the sequential paradigm implied by von Neumann.

Slow Network Speed Growth

For a long time now, the speed of all communication layers (not only the Internet, but also on-chip databus) is growing much more slowly than the consumption rates of processors and processing units. Although the clock speed isn't increasing, the number of consumers (cores, processors, and devices) continues to grow. As the communication speed recedes, computation becomes increasingly faster than data access, leading to delays in execution.

Impact on the Traditional Model

These modern requirements contradict the traditional approaches as laid out by von Neumann and Turing.

Parallelization

With the requirement to compensate performance by number of instances, the primary principles already fail. Although the function is essentially still encoded in a sequence of operations, dependencies between operations and shared data are introduced. This means that the actual work to be performed can no longer be considered a (single) sequence of actions, or that code and data are necessarily colocated. Work that's enacted in parallel differs conceptually from work performed sequentially. It doesn't always require direct coordination and communication between actors.

From the perspective of the traditional models, this means that operations are collectively executed over a shared information resource. This implies that the execution context can be local and comparatively large, but it isn't static or restricted to a single access. Most natural parallel systems are able to deal with errors, rather than being strictly accurate—even on molecular level.

Increase of Data

The original computing systems were expected to act on a very limited scope of data that could easily fit in the processing unit storage. As performance increased, so did the need to process more data, which soon exceeded the available storage and posed physical problems in terms of speed (fast memory is expensive, is limited in scale, and should be physically close to the processing unit). This gave rise to storage hierarchies with faster, smaller memory colocated, and slower, larger memory located further away. This distribution reduced the impact of access delay considerably.

Because modern applications exceed even this range, new ways of operating on data need to be considered. By nature, data is no longer colocated to the executing instance and access to it is no longer immediate. Instead, execution performance becomes more and more data driven, implying that the code must take a data perspective and allow for operations independent of context, location, and/or data in an asynchronous fashion.

The Turing model implies completeness of context and operation at all times, not allowing for partial execution, or inconsistent or incomplete state. Operations act over a stream of data, rather than the other way around, which would release the operation from its context.

Releasing Colocation

Following closely on the size problem, with mobility of the processing units, data locality is no longer static. Not only does the data size exceed storage and therefore the data moves further away from the processing unit, the modern unit itself has become mobile, moving further away from the data. Some modern cloud solutions "drag" the data behind the device, by moving it to a closer colocation, once the new location of the device is known. Because the network speed is slower than the travelling speed of mobile devices, this further impacts performance speed.

Although for von Neumann and Turing data is colocated and access is immediate (see Figure 2), these models don't consider the concept of having to maintain remote memory and waiting for availability. Out-of-order execution and pipelining are early attempts to compensate for this problem to some degree, by exploiting the fact that some operations are fairly independent of each other and therefore can principally be executed at the same time. But the gain of this approach is constrained to a few cycles, as opposed to the milliseconds and even seconds needed in modern environments.

Multitenancy

Not only is data no longer colocated, it's also shared between instances in different locations. This goes far beyond the problems already being incurred through parallelism, because it reduces the degree of controllability considerably (multiple processes with different behaviour can manipulate the data) and leads to higher access delays. As a consequence, data consistency and therefore "correctness" can't be guaranteed. As before, this implies that the code and data relationship needs to be relaxed beyond what von Neumann foresaw.

The problem, however, isn't only related to the processor architecture, but also to the (Turing) execution model, which implies correctness at all times. More advanced programming concepts must find mechanisms to translate the problem into a more relaxed algorithm.

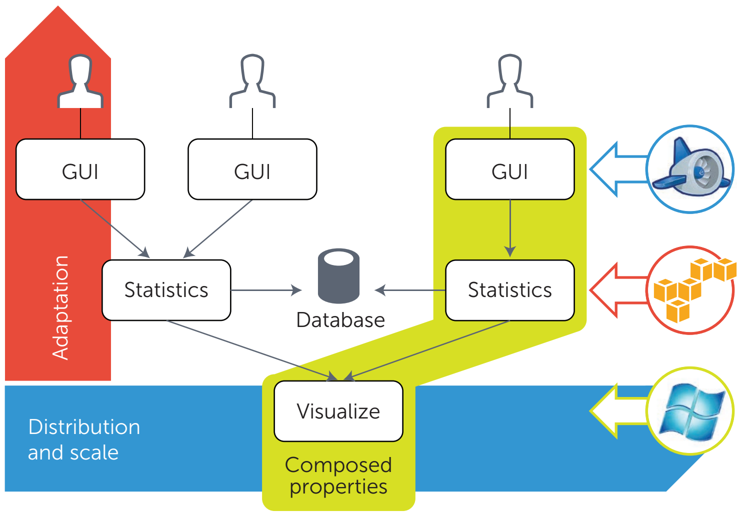


FIGURE 3. By modularizing an application into a sequence of functions with operational and data dependency, different scaling and distribution behaviors that better exploit the infrastructure can be achieved.[3]

Current Approaches

These problems aren't new to IT and in fact it can be easily shown that ever since the advent of computers and networks, a primary development goal has consisted of being able to deal with the distributed nature of the processing unit and data, as well as the implications of sharing and parallelization. What's more surprising is that most attempts to overcome the inherent problems of the traditional model were constrained to building complex functions on top, rather than trying to come up with an alternative.

This doesn't mean that the modern community and industry is unaware of the underlying paradigm shift. Almost all IT areas are working hard to address parallelism, integrating remote data sources, supporting mobility and adaptation, and so on. However, they approach the problem through complicated mechanisms on top of traditional methods.

Modular Programming

A general principle noticeable in all approaches consists of merging complex operations into a single function and exposing this as the new atomic unit. In a modular programming approach, software is developed as a sequence of operations over these functions—that is, the functions are modular subtasks of an overarching workflow. The difference from traditional models is due less to this setup than that modular structured applications can principally be managed as distributed, scalable programs. Projects such as PaaSage (www.paasage.eu) investigate exactly this manageability and how to use it for dynamic program execution on demand.

In this model, functional elements are connected primarily by their need to exchange information, without specifying concrete timing. This allows principally independent treatment of the modules, up to the point where data is expected to be shared. If carefully developed, however, the modules aren't dependent on the most up-to-date instance of the data, but allow the application to act on cached and semiconsistent information.

Dependencies are thereby primarily bound to nonfunctional properties underlying the application purpose, including aspects such as actuality of data and response time. These factors, rather than anything else, define the behavior of the applications, their scalability, and so on (see Figure 3). The modules are thereby secondary and can be replaced with functionally equivalent modules, according to the principle and specification.

However nice this is in theory, it doesn't quite work like this in realistic scenarios: in reality, services are unique and matching them against each other has proven to be a task the semantic Web is still struggling with. Repurposing modules on the basis of their intentions is therefore still a major problem, as is composition and decomposition.

Self-Adaptive Systems

Following the principles of natural systems, in which elements in a swarm adapt according to very simple principles, such as "follow the bird before you," and

```
for (int i=0; i<SIZE; i++)
        a[i] = a[i]/n;

workflow:
        a[0] -> a[1] -> a[2] -> ...

dataflow:
        a[0] || a[1] || a[2] || ...
```

FIGURE 4. A simple normalization function can be depicted as a work- or dataflow. It is obvious that while the workflow seems to prescribe an order, such an order does not exist from the actual data usage point of view.

without requiring full coordination and communication, self-adaptive models are essentially the next generation of multiagent systems. They try to shift the adaptation logic from a strict predefinition of different modules into functions that manage to configure themselves without fixed predefinition.

The principle, however, requires either a fully adaptive programming model or just tries to cover up the underlying correctness and strictness of the Turing model. This makes programming more complex than anything else, as the total function depends on the immersive behavior of the individual modules.

Dataflow-Oriented Programming

Because communication and data access are the primary performance bottlenecks for modern applications, a more data-centric view on application execution is required. One of the first examples of this approach is the Manchester dataflow machine in the 1970s.

The Turing and von Neumann concepts basically saw workflows as a depiction of the algorithm's behavior, that is, a predefined sequence of operations to be executed to achieve the results. As opposed to this, the dataflow definition investigates how data is transformed by a set of predefined functions, and the operational order derives from the evaluation of the function, respectively, from the data-specific requirements of each.

Figure 4 illustrates the difference between workflow and dataflow on a simple normalization function. Because the operations on each element are independent of each other, the dataflow exhibits a clear parallelism of the work, whereas the workflow implies an (unnecessary) sequentiality of the execution. This gives dataflows a far higher potential for parallelism, but also for scheduling, prefetching, and so on.

Programming extensions such as StarSs[4] try to add data dependency information so as to exploit parallelism, execute tasks out of order, and so on, thereby principally allowing the application's execution environment to exploit data location (see Figure 5).

To fully exploit dataflow information, however, an execution engine is necessary to constantly supervise task readiness. As a consequence of such an engine, the individual tasks in the dataflow need to be large and complex enough to compensate for the management overhead. Tasks, obviously, still follow the traditional model themselves, thus ranking (current) dataflow approaches close to modular programming.

Functional Programming

Functional programming takes a mathematical approach to defining a problem, as opposed to the imperative Turing concepts. Because the mathematical description doesn't prescribe a specific execution order (other than the evaluation constraints) or algorithm, it offers the maximum freedom to exploit parallelism, out-of-order execution, and so on.

By nature of a mathematic declaration, the same formula can even be converted using basic mathematical theorems (see Figure 6). The POLCA project (www.polca-project.eu) uses this principle to change the dataflow, aggregate and decompose functionalities, and so on, to alter the complete application behavior, and thus allows the compiler to affect parallelism, exploit hardware characteristics, and so on.[6]

The mismatch between the Turing model and the mathematical description can lead to major performance losses if the algorithm isn't specifically optimized for von Neumann architectures. Similarly, functional descriptions foresee no concept of data access penalty either, nor do they even bring a notion of memory. Functional languages therefore run the risk of adding a similar access wait time than imperative languages and could result in memory overflow. The POLCA project, for example, takes a stance between functional and modular programming. It should be noted that many of the notions of functional programming, such as lambda operators, find increasing interest even in the imperative programing.

Rethinking Software: The Triple-I Model

The European Commission recently released a report based on a discussion among approximately 100 experts from industry and academia on how to address the challenges of future IT.[7] This report proposes the "Information-Incentive-Intention" model as a basis for "a shift from Turing and von Neumann

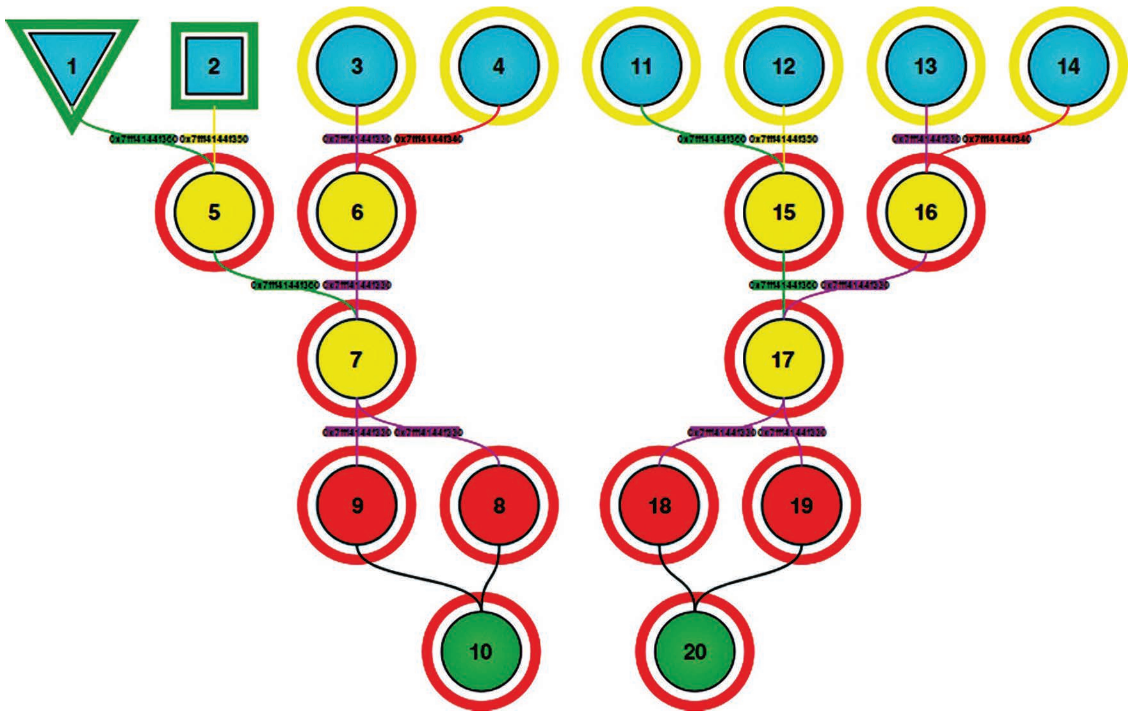


FIGURE 5. Using the StarSs programming model, the compiler can generate a dependency graph of all data shared between functions. This dependency graph allows scheduling according to data availability at runtime.[4,5]

towards a new ICT of distributed parallel information valorization."[8]

Information-Incentive-Intention

The relationship between execution and code/data structure constrains the program's adaptability, representation, and usability. It also constrains how software is executed, data is placed, and distribution is managed. By building a strict one-to-one representation between consumed data and bytes in storage, and between algorithm and operations to be executed, applications are bound to their platform and use. What's more, they're difficult to maintain, adapt, and debug.

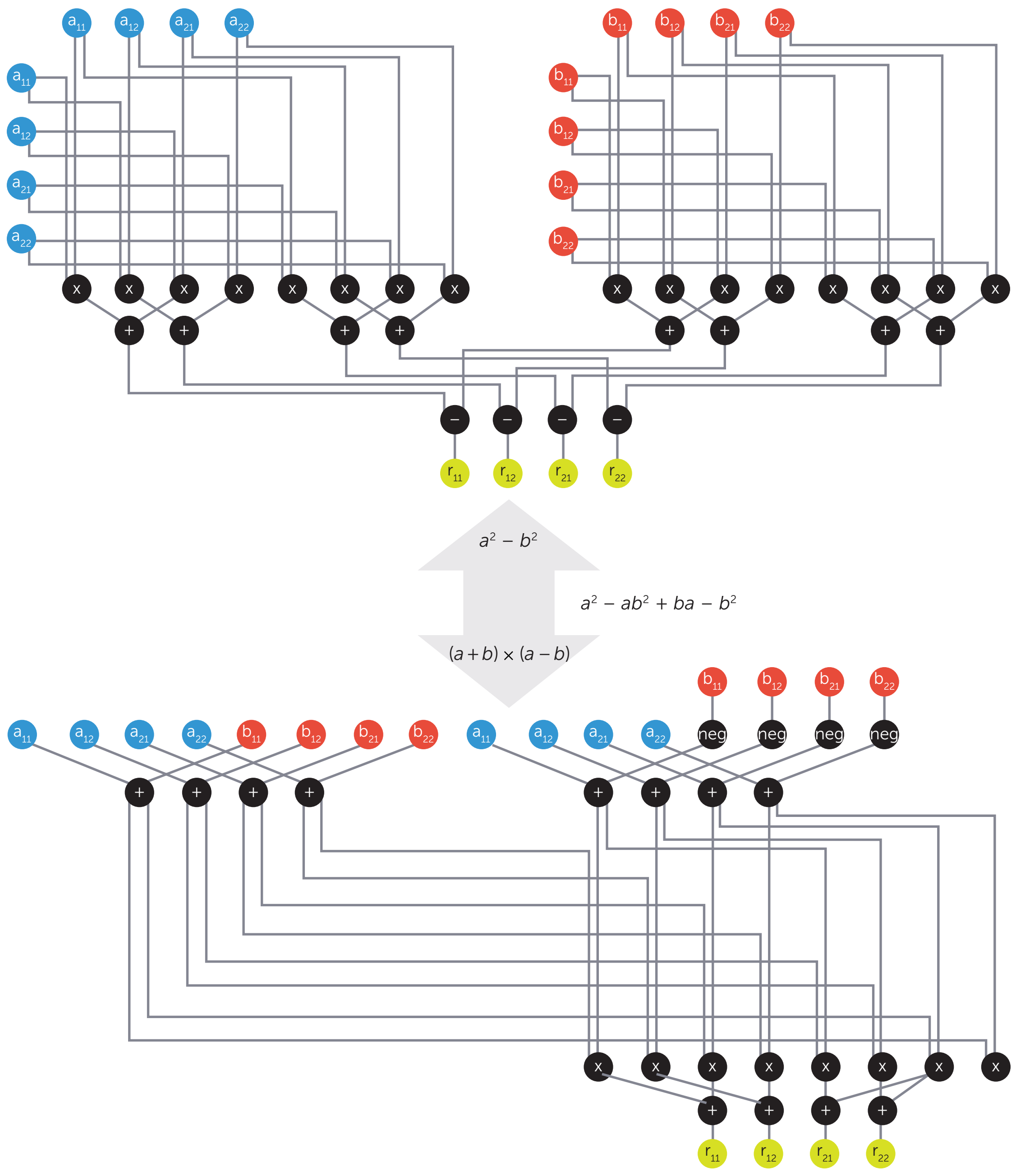
It's therefore necessary to think of code and data in the form they're actually intended to be used in and the information they carry. This follows the principles of an increasing abstraction level, such as from assembly to C. Abstraction is only sensible if the technical constraints are considered, and it's already difficult to efficiently match the current level of abstraction, let alone increase it further.

Bearing the development and requirements in mind, this requires developing a model with the aim of fulfilling the triple-I abstraction, which is illustrated in Figure 7.

Information: Abstracting data. Even though we must develop software to adhere to the bit encoding, developers don't have to have a specific data structure and operations on it in mind when designing the application. Instead, all applications serve the purpose of processing (consuming and providing) information.

Information is inherently different from "meaning" and must allow full reconstruction of all relevant data but in any format thinkable. This implies some format of convention and standardization (such as ASCII) to maintain the unchangeable data, but in particular and more importantly, some form of metadata (even, potentially, of a semantic nature) that is itself adaptable and represents the different interpretations and hence representations of the data.

The main reasons to represent data as information are to guarantee its transformability into different views and usages, and to enable its composition and decomposition, such as for information mining or to communicate and act on partial data. As with adaptive bitrate streaming, functions that have access to only incomplete data should be able to perform partial processing on the basis of the information available.



a_{11} a_{12} a_{21} a_{22}
b_{11} b_{12} b_{21} b_{22}
r_{11} r_{12} r_{21} r_{22}
$a^2 - b^2$
$a^2 - ab^2 + ba - b^2$
$(a + b) \times (a - b)$
neg

FIGURE 6. Two mathematically equivalent functions will, just by nature of their definition, expose completely different data-flow models, if converted into an according algorithm. By exploiting mathematical theorems, such functions can be transformed from one dataflow form into another.

Information is a partial representation of the data, in the sense that it provides a metadata view on the concrete bitwise encoded data, whereby data is the most concrete information and metadata the most abstract view.

Intention: Abstracting code. A similar degree and type of abstraction applies obviously to code itself. Not only must the code be able to act on information (including incomplete information) rather than data, it must also be able to become transformable, adaptive, and (re)composable itself.

Traditional implementations have the code optimized for the specific usage context, respecting in particular the hardware characteristics. As the scope of devices increases, so must the code's portability, without constant recompilation. To remain flexible and dynamically instantiatable, the code must adapt on the fly. Moreover, if the same application is used in different contexts, its requirements and dependencies in relation to other services will change each time.

We must distinguish between definition of the code (its purpose or intention) and its actual execution. The relationship between these two must allow for the necessary performance, and thereby implies that the mapping involves not too many intermediary steps. Interpretation on virtual machines is thus certainly not a sustainable solution.

A noteworthy example is the "foreach" construct, which carries the information and intention that the subsequent code can be executed independently in any order, and yet still is easily understandable and usable.

By nature of the intention and the capability to repurpose and recompose applications out of different intentions and information, intentions can be recursively combined and applied. As such, the intention of an individual function must not necessarily comply with the intention of the overarching application or use case.

Incentive: Ensure goal maintenance. Intentions can only be adhered to at a certain level of quality: almost any application and any data can be repurposed for different contexts if performance criteria are ignored. We define, as an "incentive" of the execution, which of these properties need to be maintained and how well. The incentive behind an application is thus related to its business goal.

As the capabilities to recompose and repurpose services grow, so does the complexity of translating such requirements into concrete properties. Because such compliance is business essential (such as producing unpredictable cost), the relationship to the composed application must be well-defined and executable.

This implies understanding the intrinsic immersive properties of a complex distributed system where the individual components collectively expose specific properties. This is basically an unsolvable task (finding the desired solution for an underspecified equation system) and must be supported by heuristics, hidden limiters, inaccuracy compensation, and so on.

Realizing Triple-I: A Roadmap

Similar concepts to those presented here guide many research and development efforts in software engineering. The main obstacle consists in their

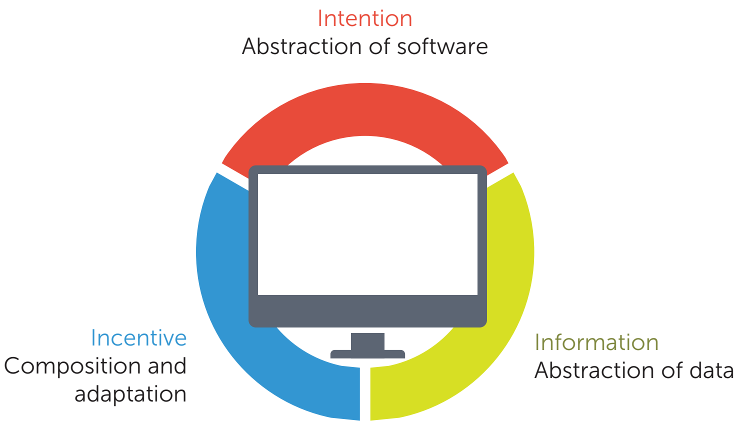


FIGURE 7. A new software paradigm that manages to overcome von Neumann and Turing must cater to Information (instead of data), Incentive (instead of imperative declaration), and Intention (instead of relying on the operating system's intelligence).[7]

realization, as the concepts are far from the actual execution model and thus not applicable without an implicit loss in performance due to this mismatch. This is why a complete rethinking of the traditional software model (programming and execution) needs to take place.

Economic concerns, however, clearly reject such an approach, as it would undoubtedly reset all of our IT capabilities by decades: millions of services, applications, industry areas, hardware, and so on rely on the current model, and a completely new model would essentially void all this progress.

Next to the fundamental change comes the need to find a smooth transition from the current industrial standard in a fashion that's economically feasible, competitive, and sustainable. This means that at the same time that the foundational capabilities are reexamined and a new IT is built, preparatory actions must be taken to move the current IT toward this new model. This implies two (interleaving) roadmaps: one of the foundational research and one of the actually used models and their transition.

The cloud computing expert group and the software engineering experts thereby came up with the following roadmap that tries to exploit a maximum overlap between foundational research advances and technical transition—that is, the same milestones can be used in both cases (see Figure 8), yet with an obvious difference in scope and concreteness.[7]

The goal of the *information* branch is to steer the development from concrete data toward abstract structures that can be transformed into different concrete data representing different views on subsets of the information:

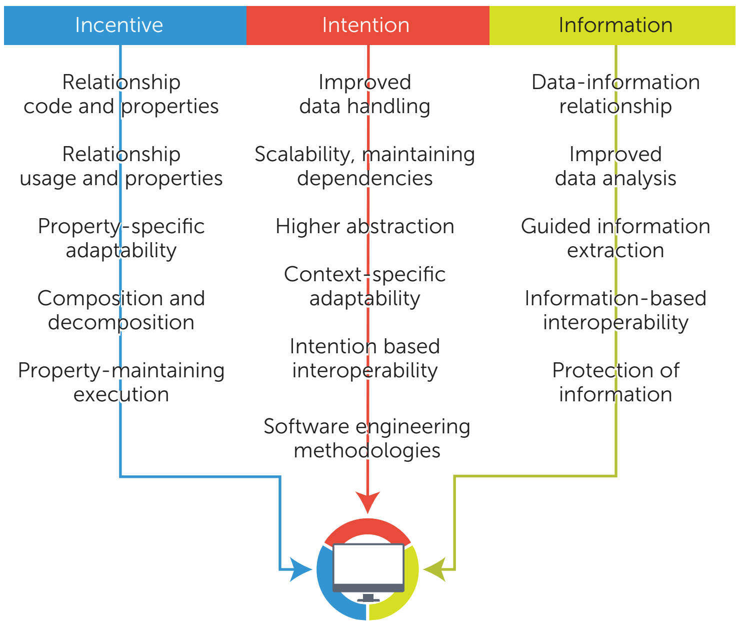

Incentive
Intention
Information
Relationship code and properties
Relationship usage and properties
Property-specific adaptability
Composition and decomposition
Property-maintaining execution
Improved data handling
Scalability, maintaining dependencies
Higher abstraction
Context-specific adaptability
Intention based interoperability
Software engineering methodologies
Data-information relationship
Improved data analysis
Guided information extraction
Information-based interoperability
Protection of information

FIGURE 8. The software engineering roadmap as promoted by the European Commission Cloud Computing Expert Group.[7] The roadmap consists of three branches—information, intention, incentive—each with distinct goals.

- *Investigate the relationship between data and information*: Data currently tries to be a one-to-one depiction of reality. This also affects how we represent information—by using exact data. Simulations typically try to reproduce reality, rather than extract the actual information of interest.
- *Information mining*: Data analysis and data mining are about extracting information from big datasets. The full potential is only achieved once the analysis acts directly on the information itself.
- *Information-guided functions*: These functions deal with and aim at specific types of information. This would, for example, enable goal-oriented queries over large datasets.
- *Information interoperability*: Interoperability implies not only that information can be extracted from data and that information can be understood, but also that equivalent datasets can be generated from information.
- *Information protection*: Data is no longer the source of privacy concerns, but the information gained from data. With the power of information processing increasing, new ways to protect information and individuals are required.

The *intention* behind a piece of software is still inaccessible for any compiler and even unintelligible for most human readers. The complexity of cross-referencing, pointers, function invocations, and so on makes code generally unreadable, unless it's commented on accordingly—that is, unless additional information on the intention is provided. This branch aims to make such an intention a more integral part of the programming model and software engineering processes:

- Understand the relationship between intention (such as a mathematical formula) and the different algorithmic implementation/execution choices. This includes how data is handled and transformed, as well as which intention can be expressed algorithmically with which effects.
- Because distribution, scaling, and sharing substantially change the application behavior and, in particular, its characteristics, define applications such that their behavior can be defined on the basis of its implicit characteristics, rather than having to explicitly encode any potential occurrence.
- Build a syntax that's expressive enough and at the same time carries all the information relevant for compilation and execution.
- Develop the ability to transform the different code expressions in a controlled fashion, just as with data. In other words, intentionally equivalent (pursuing the same goals) code can be generated from the same intention declaration. Figure 9 is a simple example.
- Adapt code (component, module) to a specific usage context, such as the device it is to run on. This applies on the lowest level of granularity, as more complex functions are actually compositions of individual low-level modules.

Incentive is immediately reminiscent of intention. However, whereas the intention of a code reflects the behavioral goal of the application, its incentive defines the properties it needs to fulfill during execution, including user satisfaction:

- Formal analysis of the relationship between a given code and the properties it exposes; in particular, properties related to distribution, scaling, and sharing have only been insufficiently analyzed so far.
- Impact of usage on these properties. Such factors need to be formally assignable to application characteristics so as to assess when to change the behavior, and vice versa, in order to assess to which degree such properties can actually be fulfilled during execution. Performance and re-

lated characteristics still won't be accurately assessable, though (the halting problem). Instead, it's expected that more general predictions (such as whether hosting the data somewhere else is likely to increase response time) can be made.
- Ability to transform code and data to meet different nonfunctional objectives (such as quality of experience).
- Bring all of these advances together to realize composition and decomposition of functionalities, characteristics, and properties. This requires not only a full scope of transformation capabilities, but also a full understanding of all relationships of all listed aspects between each other. Transformations thereby imply composition and decomposition of functions and their properties.
- Execute software that maintains the essential incentives (such as business goals) in a distributed, shared, and dynamic environment.

All three factors together—information, intention, and incentive—form the requirements that a new software engineering model has to respect.

It's easy to see how the promoted approach differs from the standard Turing/von Neumann–based model. The steps laid out in this article suggest that we promote an intelligent system following on semantics and multiagent system (MAS) development. This, however, isn't so. In fact, MAS and semantic approaches so far have generally failed to meet the expected performance and behavior because many of the factors here (such as the relationship between code and properties) can't be formally specified in the von Neumann context.

The steps described aren't suggestions for higher level functionalities to be added to the current IT paradigms, but are to be taken as design principles for developing a new paradigm from the ground up. This means that the capabilities need to be directly supported by the hardware and ISA. This doesn't mean that the ISA should support intentions, but that it provides the necessary instructions that make intention in the form we've described possible in the first instance.

This equally implies changes on the hardware as well as on the software side and necessitates an intensified codesign in the first instance. The amount of effort needed versus the direct benefit gained makes changes in the IT paradigm unattractive to industry, although the need for it will only increase, as most companies are well aware of.

```
"intention" , with c const.

code 1:
double f( double x )
{            return c*x; }

code 2:
double f( double x )
{            double res=0;
             for (int i=0; i<c; i++)
                          res+=x;
             return res;
}
```

FIGURE 9. The same "intention" of a code (that is, the result it should generate) can be expressed in various algorithms. Here, code 1 and code 2 are "intentionally" equivalent, though their behavior and performance will obviously vary.

The roadmap promoted here can be enacted on the iterative and the foundational levels, whereas the iterative route level primarily generates knowledge and allows a direct comparative analysis of different approaches, and the foundational level requires a considerable amount of rethinking the essential theories (such as how to access data and how to execute operations).

Acknowledgments

This work was partially supported by the POLCA project under FP7-ICT-2013-10, grant agreement no. 610686, and the PaaSage project under FP7-ICT-2011-8, grant agreement 317715. Further thanks go to the members of the European Commission Cloud Computing Expert Group and the Software Engineering Expert Group. This work was also prepared in strong collaboration with HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation.

References

1. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014–2019*, white paper, Cisco, 2015, www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.
2. F. Richter, "Social Networking Is the No. 1 Online Activity in the US," Statista, 2013; www.statista.com/chart/1238/digital-media-use-in-the-us.
3. K. Jeffery, G. Horn, and L. Schubert, "A Vision

for Better Cloud Applications," *Proc. 2013 Int'l Workshop Multi-cloud Applications and Federated Clouds*, 2013, pp. 7–12.
4. J. Planas et al., "Hierarchical Task-Based Programming with StarSs," *Int'l J. High Performance Computing Applications*, vol. 23, no. 3, 2009.
5. S. Brinkmann et al., "TEMANEJO: A Debugger for Task Based Parallel Programming Models," *Proc. Int'l Conf. Parallel Computing* (ParCO 11), 2011.
6. L. Schubert, J. Kuper, and J. Gracia, "POLCA: A Programming Model for Large Scale, Strongly Heterogeneous Infrastructures," *Proc. Int'l Conf. Parallel Computing* (ParCO 14), 2014.
7. K. Jeffery and L. Schubert, "Complete Computing: Toward Information, Incentive and Intention," European Commission, 2014; http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm? action=display&doc_id=6775.
8. L. Schubert and K. Jeffery, "Challenges in Software Engineering, H2020: Analysis and Summary of the Cloud Expert Group Reports," European Commission, 2014; https://ec.europa.eu/digital-agenda/events/cf/cloud-computing-software-engineering/item-display.cfm?id=14093.

LUTZ SCHUBERT *is head of department at the Institute for Information Resource Management, University of Ulm, where he leads research on high-performance computing and cloud provisioning concerns. His research interests include parallel heterogeneous infrastructures and their management and programmability. Schubert has a diploma in computer science and philosophy from the University of Stuttgart. Together with Keith Jeffery, he acts as moderator for the European Commission Cloud Computing expert working group. Contact him at lutz.schubert@uni-ulm.de.*

KEITH JEFFERY *is an independent consultant with Keith G. Jeffery Consultants and past director of IT at STFC Rutherford Appleton Laboratory. His research interests include information systems architecture, interoperability, virtual research environments, grids, clouds, and metadata. Jeffery has a PhD in geology (with a large IT content) from the University of Exeter. He holds three honorary visiting professorships, is a fellow of the Geological Society of London and the British Computer Society, a chartered engineer and chartered IT professional, and an honorary fellow of the Irish Computer Society. Contact him at keith.jeffery@keithgjefferyconsultants.co.uk.*