# Knowledge Management for Inclusive System Evolution

Tiziana Margaria(✉)

Lero and Chair of Software Systems, University of Limerick, Limerick, Ireland
tiziana.margaria@lero.ie

**Abstract.** When systems evolve in today's complex, connected, and heterogeneous IT landscapes, waves of change ripple in every direction. Sometimes a change mandates other changes elsewhere, very often it is needed and opportune to check that a change indeed has no effects, or maybe only the announced effects, on other portions of the connected landscape, and impacts are often assessable only or also by expert professionals distinct from IT professionals. In this paper, we discuss the state of affairs with the current practice of software design, and examine it from the point of view of the adequacy of knowledge management and change enactment in a co-creation environment, as it is predicated and practiced by modern agile and lean IT development approaches, and in software ecosystems. True and functioning inclusion of non-IT stakeholders on equal terms, in our opinion, hinges on adequate, i.e., accessible and understandable, representation and management of knowledge about the system under development along the entire toolchain of design, development, and maintenance.

## 1 Change Is Not Always Just Change

As we have observed at the International Symposium On Leveraging Applications (ISoLA) conference over the course of its 12 years and seven occurrences, research and adoption of new technologies, design principles, and tools in the software design area at large happen, but at a different pace and with different enthusiasm in different domains.

While the internet-dominated branches have adopted apps, the cloud, and thinking in collaborative design and viral distribution models, counting among the enthusiasts those who make a business out of innovation, other markets have adopted novelties either forced by need, like the transportation-related sectors, or by law, like the US government mandated higher auditing standards related to Sarbanes-Oxley, and the financial sector.

Other areas have readily adopted hardware innovation but have tried to deny, resist, and otherwise oppose software-driven agility deriving from the internet economy. An illustrative example is the traditional telecommunication companies, cornered in several markets by the technology that came together with the social networks wave.


B. Steffen (Ed.): Transactions on FoMaC I, LNCS 9960, pp. 7–21, 2016.
DOI: 10.1007/978-3-319-46508-1_2

Still others are undecided on what to do but for different reasons; mostly they are stuck into oligopolies such as in the ERP and business information system-related enterprise management software industry. These oligopolies set the pace of adoption: they try on one hand to slowdown change in order to protect their old products and on the other hand they try with little success to jump on the internet wagon but failing repeatedly. A prominent example is SAP with their "On demand" offers, whose development was recently stopped.

One other cause of current indecision, as prominently found in the healthcare industry, is an unhealthy combination of concurring factors which include:

- the *fragmentation* of the software and IT market where big players occupy significant cornerstones,
- the *cultural distance* between the care providers (doctors, nurses, therapists, chemists...) and care managers (hospital administrators, payers, and politicians) from the IT professionals and their way of thinking in general and the software design leading-edge reality in particular,
- and the *hyperregulation* in several undercorrelated layers of responsibility by laws as well as by professional organizations and other networks that represent and defend specific interests of niche actors.

## 2 Change at Large

In this very diverse context, we see that change in these different domains faces an amazing diversification of challenges. These challenges are never addressed in simple, rational decisions of adoption based on factual measurements of improvement of some metric such as cost, efficiency, or performance as in the ideal world every engineer fancies. Instead, they involve complex processes of a socio-technical character, where a wealth of organizational layers and both short and long term interests must be aligned in order to create a sufficiently scoped backing to some envisaged change management measure.

Once this is achieved (good luck with that!), there is the operationalization problem of facing the concrete individuals that need to be retrained, the concrete systems that need to be overhauled (or substituted), and the concrete ecosystem surrounding these islands of change: they need to support or at least tolerate the intrusion of novelty in a possibly graceful way.

Seen this way, it does not surprise anymore that the success rate of software projects stagnates in the low double digit quartile. It is also clear that the pure technical prowess of the staff and team, while a necessary condition for success, is by far not sufficient to achieve a successful end of a new IT-related project.

While we, within IT, are used to smirk at cartoons that depict traits of the inner software development teams facets[1], we are not used to look at ourselves from the "outside". For example from the point of view of those other professionals we de facto closely work with, and whose professional and societal life we

[1] For instance, in the famous "tree-swing-comic". Here [3] you find a commented version and also its derivation history, dating back to the '70s.

directly or indirectly influence - to the point of true domination[2]. There is hardly anything today that is not software-impacted, and as Arend Rensink writes [27], software is never finished.

I use to tell my students that to create a piece of software is like getting a baby. First of all, it takes a couple for it: IT and those who (also) wish this piece of software and provide their contribution. In both cases, although technically an initial seed is technically sufficient, the outcome is better if a heterogeneous team joins forces, collaborating all the time for the health and safety along the build process, and preparing a welcoming environment for the new system in its future home. While we in Software Development care a lot about the (9 months equivalent of the) creation time and what can go wrong during this relatively short building period spent in the lab, we hardly practice as much care and foresight as optimal parents do for the delivery itself and its acclimation in the operational environment. This phase is called installation or implementation, depending on whether you are a software engineer or an information system person, resp. In addition, we fail quite miserably short when considering the subsequent 18–21 years of nurturing, care, and responsibility. Concerning maintenance and product evolution, in fact, our practice is still trailing the care and dedication of good parents. The Manifesto of FoMaC [29] addresses in fact this phase, and what can be done before in order to face it in the most adequate and informed fashion.

As for parents, once the baby is born and brought home, one is not done and able to move to the next project, but the real fun begins. There is a long time of nurturing and care and responsibility, that in theory is 18–21 years long. Funnily, this is not so distant from the age of maturity of most large software systems, and it is usual to find installations that are even older than this, especially for core modules in business critical platforms. In reality, however, the bond of care never fades completely. If we were parents, with our current "best practices" in IT, we would be utterly careless, and in several nations we would have the youth protection offices on our door, determined to put our children in foster care.

So, what can we do?

## 3 Change and Simplicity

A first step would be to improve our awareness of the impact radius of our products (software, hardware, communication infrastructure, and the systems in which all the above is embedded or that use it as part of their operating environment), artefacts (models, code, test cases, reports, documentation, specifications, bits and pieces of knowledge formulated in text, diagrams, constraints, ontologies, and scribbled notes from innumerable meetings), and professional decisions. This cartoon brought me to think: As long as "the world" sees the IT professionals as extraterrestrial geeks who speak some incomprehensible lingo, there is a deep disconnect that is worse than the already abundantly demonized but still unsolved "business-to-IT" gap. We are not a cohort of Sheldons (from The Big Bang Theory), nor Morks (of the unforgettable Mork and Mindy),

[2] This is genuine bidirectional incomprehension: as aptly captured in [1].

nor wizards that dominate machines with some impenetrable arts and crafts as in Harry Potter's world. As engineers, we must become understandable to the world. In this direction, there are three actions to be taken: simplify, simplify, simplify.

- Dramatically simplify the **domain knowledge gathering** process: requirements and specification come from the field, far out there where no IT culture is present. We depend on "them" to produce what they need, and it is our responsibility to go all the way down to their homes and make their ways and languages understandable to us.
- Simplify **the way we design systems**. It can no longer be that we have n different kinds of memories, m layers of middleware between the firmware on one side and more layers of software on the other side. This is too much inherent diversity that leads to hardships in keeping coherence and faithfulness between the different representations, interpretations, and implementations in a world where every few weeks there is a patch, every few months a new version or a new release, and this for every component in the assembly of systems we face
- Simplify the way we take **decisions about the evolution** of systems. This decision making needs to be tight between the domain experts, who define the end product, its features, and express desires driven by the application's side needs, and the IT team that manages the technical artifacts that describe, implement, and document the current status, the next version, and a feasible design and migration path between the two. Repeat forever.

This is the central scope of FoMaC, and these different aspects are its itemized workprogramme, only recast in terms of simplicity as a core value.

## 4 Simplifying Knowledge Management

For a functioning handshake between business and IT, knowledge sharing and trading across cultural borders and barriers is a must. As a convinced formal methodist, I firmly believe in the power of materializing rules, regulations, preferences, best practices, how-to's, do's and don'ts in terms of properties, expressed as constraints. Such constraints are materialized in some logic in order to be able to reason about them, and the choice of which logic depends on the nature of the properties and on the technical means for working with those properties. Knowledge, like energy, can assume different forms, and become useful for different purposes. There is a cost of transformation that includes the IT needed to "manage" the transformation, in terms of data mediation across different formats, compilers across different representations (that for me include programming and process languages), or preparations for being more universally accessible, as in ontologies and Linked Open Data. Taking a constraints view at knowledge, facts ("things", concepts, and data are facts) are just constants, properties are unary predicates, and n-ary predicates formalise multiparty relations, including relations about constants and other relations.

The conceptualization and materialization of knowledge in formally well defined terms that can be analyzed by tools is an essential step towards knowledge exchange, use, and even growth. How can we know that something is new if we don't know what is known today? In different flavours, this question pops up in many application domains: what is evidence-based medicine, if we cannot describe and evaluate the fit of the available evidence to the case for which a doctor submits a query? The knowledge and evaluation of the plausibility, up-to-dateness, and overall fit of background evidence knowledge is of paramount importance for its applicability to the extrapolation to the single subject: evidence gathered in the past, in unclear experimental circumstances, from a different population basis, might not be applicable at all, and even lead to wrong diagnosis or treatment. For example, the fact that there are biases in the wealth of widely accessible medical experimental data is well known to pediatricians or gynecologists, who constantly need to treat children or (pregnant) women resorting to assessed evidence mostly collected from experience with adult male patients - experiments with "at risk" groups are extremely rare and very difficult to achieve ethical approval. Indeed, even the processes of paediatric first aid in Emergency Medicine are not defined, and we are working right now with the European Society of Emergency Medicine to model a variant of the adult processes. This model materializes the knowledge of the expert EM paediatricians, exposing their practices and beliefs for the first time to a methodological discussion that is amenable to further machine-based reasoning. Properties can be declarative, describing abstract traits of a domain or thing or process, or also be very concrete, if applied to single elements of the domain of choice. In terms of knowledge management, subject matter experts usually expect IT specialists to master the knowledge description and its corresponding manipulation, for several reasons:

- knowledge is usually expressed in difficult ways, not accessible to non-IT specialists, like logics formulas and constraint or programming languages. Also ontologies and their languages of expression, that in theory should make semantics and concept description easy for everyone, are still for initiated.
- manipulating knowledge concerns handling complex structures, and is mostly carried out at the programming level. Also query languages are not easy, especially in the semantic domain: description logics and W-SML or SPARQL are not easy to master.
- knowledge evolution and sharing by normal practitioners function today only in a non-formal context, mostly textual: they are able to update and manage Wikis for communities, access repositories of abstracts and papers, but it is not common for subject matter experts with little or no IT proficiency to go beyond this on their own. Few of the target people master spreadsheets (which are at best semistructured and primitively typed repositories), and far fewer are proficient in database use and management.

The key to enlarging the participation to knowledge definition and management lies therefore in a system design that fosters bridges into the IT world,

instead of creating generation after generation of IT walls and canyons. Ideally, it should start with the high level requirements and continue coherently throughout the entire lifetime of the system and its components.

## 5 Simplifying System Design

In the IT industry, system design today still happens mostly at the code level, with some application domains adopting model driven design in UML, SysML or SimuLink style. Other application domains, mostly in the business sector, are gradually embracing an agile development paradigm that shortcuts the modelling phase and promotes instead code-based prototyping.

### 5.1 The Issues with Code

The tendency to directly code may have immediate advantages for the IT professionals that right now write the code, but it tends to hinder communication with everybody else. With or without agility, code first scores consistently low on the post-hoc comprehension scala pretty much independently of the programming language. Indeed, a widespread exercise for programming newcomers is to give them a more or less large pre-coded system and ask them to make sense of it. Reverse engineering is difficult when starting from models, but gruelling if starting from code. A technical and pretty cryptic (en)coding is not an adequate mean of communication to non-programmers and fails the purpose of simplifying domain knowledge management as discussed in the previous section. To compensate, according to best practice guidelines, it should be in theory accompanied by a variety of ancillary artefacts like documentation, assertions as in JML [17], contracts [26] as in Eiffel, diagrams as in the early days of SDL[3] and MSCs in the telecommunication domain, and test cases as the de facto most widespread handcrafted subsidiary to source code. In practice, these artefacts are rarely crafted, if ever crafted they are rarely consistent, and if they are consistent at some design point, they are rarely maintained along the long lifetime of a system. Even if they were maintained consistent, a complete system description would also require information about the usage context. For example, if a platform changes, an API is deprecated, a library in the entire technology stack above or below evolves, this may impact the correct functioning of the system under design or maintenance.

In theory, such changes and their effects need to be captured in a precise context model. It is pretty safe to assume that such a context model is nonexistent for any system of decent size and complexity. Most experts would feel comfortable in stating that such a model is impossible to create and maintain due to the high volatility of this IT landscape.

Current software engineering practice has no built-in cure, it tries instead to infuse virtuous behaviour to the software engineers it forms by preaching the

[3] The Specification and Description Language (SDL), including the Message Sequence Charts (MSC), was defined by ITU-T in Recommendations Z.100 to Z.106.

importance of documentation, coding standards, test cases, annotations, etc. with little success. Unfortunately, the curricularly formed software engineers are just a fraction of the programmers employed today in industry and consultancies [12]: the large majority of the extra- or non-curricular practitioners have never received a formal programming or software engineering education, and are prone to underestimate and neglect ancillary activities aimed at enhancing correctness or comprehension.

Even if these ancillary artefacts existed, they would be difficult to use as a source of knowledge. Test cases express wanted and unwanted behaviours, and have been extensively used as source of models "from the outside" of a system. Test based model inference was, in fact, used already over 15 years ago to reconstruct knowledge about a system from its test case suite [13,14]. Automated generation of test cases is also at the core of modern efficient model learning techniques, starting from [15,20,32] to [16]. The recovered models, however, describe the behaviour of the systems at a specific level of abstraction (from the outside). They are quite technical (usually some form of automata) and, thus, not adequate for non-IT experts. Additionally, they describe only the system as-is, but do not provide clues on the why, nor on whether some modifications would still be correct or acceptable or not.

Other ancillary artefacts are closer to the properties mentioned in the previous section as a good choice for formulating knowledge. Textual descriptions and diagrams basically just verbalize or illustrate the purpose, structure, or functioning of the system therefore seem "simple" and easy to produce at first sight. However, if are not linked to a formal semantics nor the code, they are not inherently coupled with analysis techniques not with the runtime behaviour, and suffer of a detachment (a form of semantic gap) that intrinsically undermines their reliability as a source of up-to-date knowledge.

APIs in the component based design and service interfaces for the service oriented world are used as a borderline description that couples and decouples the inside and the outside of subsystems. Like a Goldilocks model, APIs and service descriptions are expected to expose enough information on the functioning of a subsystem so that an external system can use it, but shield all the internal information that is not strictly necessary for that use. We have discussed elsewhere in detail [11,18] the mismatch of this information with what a user really needs to know. In short, APIs and services expose typically programming level datatypes (e,g. integer, string) that are too coarse to enforce an application specific correctness check. Age of a patient and a purchase quantity, both commonly represented as integers, would be perfectly compatible, as well as any name with a DNA subsequence. Thus, APIs need to have additional documentation explaining how to use them (e.g., first this call, with these parameters, then this other call, with these other parameters, etc.), and the domain semantic level (attacked, so far still with little spread, by Semantic approaches, e.g. to Web services). This documentation is mostly textual, i.e. not machine readable, i.e. not automatically discoverable nor checkable. The OWL-S standard [2] had with the "process model" a built-in provision for these usage models, essentially the protocol patterns of service consumption, but it was considered a cumbersome approach, and thus has gained limited spread in practice.

Even APIs may suddenly change. Famous in our group is the day the CeBIT 2009 opened in Hannover: we had a perfectly running stable demo of a mashup that combined geographic GIS information with advanced NGI telecommunication services, plus Wikipedia, Amazon and other services... that on site on that morning partout did not work. Having excluded all the hardware, network, and communication whims that typically collude to ruin a carefully prepared grand opening, only the impossible remained to be checked: the world must have changed overnight. Indeed, the Google maps service API had changed overnight in such a way that our simple, basic service call (the first thing the demo did) did not work anymore. In other words, any application worldwide using that very basic functionality had become broken, instantly and without any notice.

Artefacts like assertions and contracts are the closest to the knowledge assets mentioned above: indeed they express as properties the rules of the game, and in fact they are typically formulated at the type level, not for specific instances. On the one side they fulfil the need of expressing knowledge in a higher-level style. On the other side, however, they are typically situated in the code and concern implementation-level properties and issues that may be too detailed to really embody domain knowledge in the way a domain expert would formulate and manipulate.

In terms of communication with subject matter experts, therefore, none of these alternatives seems to be ideal.

### 5.2 The Issues with Models

Looking at system design models from the point of view of simplicity-driven system design, we see a large spread. UML, likely the most widespread modelling language today, unifies in one uniform notation a wide selection of previously successful modelling formalisms. In its almost 20 years of existence and over 10 years as an ISO standard, it has conquered a wide range of application domains, effectively spreading the adoption of those individual formalisms well beyond their initial reach and beyond the pure object oriented design community. Particularly due to customized, profiles it has been adapted to cover specialized concepts for different publics. For example SysML, a dialect of UML for systems engineering, removes a number of software-centric constructs and instead adds diagrams for requirements, for performance, and quantitative analysis. The central weakness of UML lies in the "unified" trait: its 16 diagram types cover different aspects, sharing a notation but without integrating their semantics. Like a cubist portrait, each diagram type depicts a distinct point of view - a facet in a multifaceted description of a complex system - and the UML tools do not support the consistency of the various viewpoints nor the translation to running code. In other words, given a collection of UML diagrams of a certain number of types, there is a gap between what they describe and the (executable) complete code of this system.

Other modelling languages have different issues: specialized languages for process modelling, like BPMN [36], ARIS [28], and YAWL [34], often lack a clean and consistent formal semantic. This makes it difficult to carry out verification

and code generation, thus leaving gaps like in UML. Formalisms for distributed systems like Petri Nets and its derivatives have a clear and clean semantic, but the models become quickly large and unwieldy. Due to their closer proximity to the formal semantic model, they are less comprehensible and thus much less attractive to non-initiated.

Complex and articulated model scenarios that preserve properties and aim at code generation exist, as in [10]: they succeed in different ways to cover the modelling and expression needs of IT specialists, and to provide to the models sufficient structure and information that code generation is to a large extent possible or easily integratable. However, non-IT experts would be overwhelmed by the rich formalisms and the level of detail and precision addressed in these modelling environments.

### 5.3 Living Models for Communication Across the IT Gap

For the past 20 years we have tried, in many approximations and refinements, to achieve a modelling style and relative tool landscape that allows non-IT experts to sit at the table of system design and development, and co-create it. Called in different communities stakeholders, engaged professionals, subject area experts, subject matter experts, these are people with skills and interests different from the technical software development. In order to make a software or system design framework palatable to these adopters, three ingredients are essential:

- it should not look like IT,
- it should allow extensive automatic support for customization of look and feel,
- it should allow knowledge to be expressed in an intuitive fashion (intuitive for these people) and provide built-in correctness, analysis/verification, and code generation features.

Such a design environment should be perceived by these stakeholders as familiar, easy to use, easy to manipulate, and providing "immediate" feedback for queries and changes. In short, we need "living models", that grow and become increasingly rich and precise along the design lifecycle of a system. They need to remain accessible and understandable, morphing over time along the chain of changes brought by evolution.

**The Look.** *Living models* are first of all models, and not code. Especially with school pupils in Summer camps and users in professions distant from the abstract textuality of source code, the fact of dealing with models has proven a vital asset. Successful models are those that one can "face-lift" and customize in the look and feel to iconically depict the real thing, the person, the artefact of importance in a certain activity.

In our system design approach, the user-facing representation is highly customizable. This has happened over time and at two different levels:

- when we started, in 1994, the META-Frame Tool could be regarded as the first service-oriented environments where tool building blocks could be graphically composed into workflows [21,31,35]. already back then, META-Frame

featured a customizable look and feel of the nodes of the graphs in its graphical modelling layer, where the usual look as single or double contoured circles (as in normal DFAs, for accepting and non-accepting states), could be painted over with what today would be called sprites. So we had models in which the icons had been changed to the likeness of Calvin and Hobbes, and the entire tool was multilingual in ... for example Swabian dialect. This pliability was essential to win an industrial project with Siemens Nixdorf in 1995/96, where the IN-METAFrame tool was applied to construct a service definition environment for value-added services subsequently sold to over 30 Telecoms world-wide [5,30]. Using their own icons was back then an essential feature, central to the decision for this product against all odds. This capability is still essential today when we talk to healthcare professionals [8,19].

– today, with CINCO [33] we can customize the look and feel of the native design tools, generate the code of such tools, and customize the semantics of the "things" they design. An example of a design tool generated as a Cinco-product is DIME [7,9] for web applications, but also the Business Model Developer tool (BMT) [6] for type-safe, semantically suported, and analyzable Business Model Canvas, that are a widespread design tool in business schools. Systems are specified via **model assembly**. Here we use **orchestration** in each model, **hierarchy** for behavioural refinement, and **configuration** as composition techniques.

**Knowledge and Use.** Living models are robust towards knowledge evolution. These models are not just intuitive and likable drawings, they are formal models and thus amenable to automated analysis, verification, and, under some circumstances, synthesis, as originally described in the Lightweight Process Coordination approach [22]. The LPC approach then matured into the eXtreme Model Driven Development [23,25] and the One Thing Approach [24]. Central to all them is the ease of annotation: because they are formal models their elements can be enriched by annotations that address layers of knowledge, or concerns.

Type information is one such knowledge layer, but semantic information, annotations coming from other analyses and from external sources like ontologies or rich semantic descriptions can be attached to the nodes and edges of the graphs, and to the graphs and subgraphs themselves. Resource consumption, timing information, or any kind of property that can be seen as atomic propositions in the logic of the behavioural constraints can overlay the concrete model (e.g., quantitative for performance, qualitative for Service Level Agreements). This way, property-specific knowledge layers can be easily added to the bare graph model and allow a true 360° description, essential for the One Thing Approach: one model, but many layers of information addressing the multiplicity of knowledge and concerns of all the stakeholders.

In this modelling paradigm, layers of information corresponding to different concerns enrich the basic behavioural (functional) description, and are accessible as interpreted or non-interpreted information to the plugins. Atomic propositions are useful to the model checker. Information like execution duration, costs, etc.

are accessible to the interpreter or to various simulation environments. Compatibility information is seen by the type checker in the design environment. In the PROPHETS automated synthesis tool, structural properties of the graphs are exploited by the code generator. Knowledge, mostly in form of facts and rules used by plugins, is the central collagen that expresses different intents, different capabilities, different concerns. Consequently, it is also useful to connect different tools, different purposes, different roles and stakeholders along the design and the evolution history of a system.

Knowledge and requirements are expressed by means of properties, via constraints that are formulated in an automatically verifiable fashion. Actually, some of the constraints happen to be domain-independent, and to be already taken care of at design time of the jABC or more recently the DIME design environment. Here, for instance, this covers both the functional correctness of each model element (Action), but also the patterns of usage inside processes and workflows, like for example behavioural constraints expressed in temporal logics (typically CTL) and verified by model checking.

**The Knowledge in Use.** As our models are immediately executable, first in animation mode that proposes a walk through the system, then with real code (for simulation or for implementation), these models are enactable from the very beginning, hence the "living models" name [24] that distinguishes them from the usual software design models, which are purely descriptive and illustrative, but not "live". In most cases, such living models get refined in this style until the atomic actions get implemented, in source code or reusing previous assets like a data base, components via an API, or services. In this case, there is no inherent design/implementation gap between the initial prototype and the final product: the finished running system is co-created incrementally along the design process, and grows from the model through prototypes into the fully implemented and running system.

The execution makes the models easy to understand because it lends them a dynamic aspect that makes it possible for users that are distant from the IT design culture to "get it". I like to call this the Haptic of the design: in German, to understand is called "begreifen" which comes from "greifen", meaning "to grab", understanding comes through touch. In living models, knowledge and models are connected. Taken together, they become understandable by IT experts and Subject Matter Experts alike, bridging the cultural gap.

## 6 Simplifying Evolution Management

Managing system evolution means managing change in any ingredient of a system's well being: the needs of the user (requirements), the components or subsystems used (architecture, interfaces), the technical context (external libraries, platforms, and services), the business/legal context (regulations, standards, but also changed market needs or opportunities). Whether largely manual or largely

independent (as in autonomous systems and self-* systems), addressing evolution means having provisions in place for

1. intercepting and understanding which change this evolution step requires
2. having the means to design and express the change in a swift and correct way, validating its suitability in theory for this purpose
3. having the provisions to implement the change in the real system
4. ideally, having the possibility to backtrack and redo in case the adopted change was not fit for purpose.

The fourth condition occurs very seldom in reality, as many systems work in real time environments where there is no possibility to backtrack and undo. This makes then the second element - the validation before effecting the change - even more crucial. Indeed, formal verification methods have been pushed to success and widely adopted in the hardware circuit design industry, where design errors in the chip cannot be undone nor patched post-production.

Evolution can be seen as a transition from one condition of equilibrium (the "well functioning" steady state before the change) to another condition of equilibrium after the change. Equilibrium is seen here as the compatibility and well functioning of a hardware, software, and communication landscape that additionally properly serves the system's (business) purposes. The more the knowledge is available about the system, its purpose, and its environment, the easier it is to detect what is affected and to build transition strategies to a global configuration of equal stability and reliability.

We are convinced that it is here that the advantages of the One Thing Approach and XMDD express themselves most prominently. While at first design time they provide clarity and foster co-creation by continuous cooperation by the different knowledge and responsibility owners, in case of change it is of vital importance to be able to identify who and what is affected, what options exist within the regulatory, legal, and optimal/preferential exploration space, and then be able to execute with predictable coherence and known outcome quality.

Traditionally, change was a big deal. Architecture was the first decision, fixing the structure, often the technology, and surely the data model of the system. Essentially, architectures fixed the basic assets (hardware, software, and communications) in a very early binding fashion. This was from then on a given, an axiom too costly to overturn or amend. So it went that generations of large IT systems remained constrained and distorted by initial decisions that became obsolete and even counterproductive in the course of their life. The disruptive power of the cloud concerns firstly the dematerialization of the hardware, which is rendered pliable, and amenable to late binding, and decisions by need or as opportune. Hardware becomes a flexible part of the system design. The UI has also turned into a culturally demanded flexibility. Nowadays the digital natives wish to be mobile, on the phone, on the tablet, on the laptop. The commoditization of computing power that comes with the decline of residential computing for front-office tasks today demands GUIs that offer the same user experience on any platform, across operating systems, and possibly a smooth migration across all of them. So, after the architecture, also the user consumption paradigm cannot be anymore the defining anchor for a system design.

What remains as the core of a system or application, be it in the cloud, mobile, actually anywhere, and is the "thing" that needs to be well controlled and flexibly adapted, migrated, and evolved, is the **behaviour of the system**. The behaviour needs to be designed (and verified) once, and then brought as seamlessly as possible onto a growing variety of UIs and IT platforms, along the entire life span of usually one or more decades. Processes, workflows, and models that describe what the system does rise now in importance to the top position. In such an inherently evolving landscape, living models that are easy to design, incorporate diverse knowledge layers (also about technical issues and adaptability), are anytime executable, be it by animation, simulation, or once deployed, seem to be a very reasonable and efficient way to go.

## 7 Conclusions

The success of evolution hinges on the realization that we intrinsically depend upon the understanding and informed support and collaboration of a huge cloud of other non-IT professionals, without which no software system can be successful in the long term. As explained in this quite well known story [4], in any IT-related project, we need the right marketing, the right press releases, the right communication, the right bosses and the right customers to achieve what can only be a shared, co-owned success.

We really hope, in a few years from now, to be able to look back at this first collection of papers and see what advancements have occurred in the way we deal with change in IT, economy, and society.

## References

1. The business-it gap illustrated. http://modeling-languages.com/how-users-and-programmers-see-each-other/
2. Owl-s: Semantic markup for web services. W3C Member Submission 22 November 2004. http://www.w3.org/Submission/OWL-S/
3. The tree-swing cartoon. http://www.businessballs.com/treeswing.htm
4. The tree-swing cartoon. http://www.businessballs.com/businessballs_treeswing_pictures.htm
5. Blum, N., Magedanz, T., Kleessen, J., Margaria, T.: Enabling eXtreme model driven design of parlay X-based communications services for end-to-end multi-platform service orchestrations. In: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, pp. 240–247, June 2009
6. Boßelmann, S., Margaria, T.: Guided business modeling and analysis for business professionals. In: Pfannstiel, M.A., Rasche, C. (eds.) Service Business Model Innovation in Healthcare and Hospital Management. Springer, November 2016. ISBN 978-3-319-46411-4
7. Boßelmann, S., Frohme, M., Kopetzki, D., Lybecait, M., Naujokat, S., Neubauer, J., Wirkner, D., Zweihoff, P., Bernhard Steffen, D.: A programming-less modeling environment for web applications. In: Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016) (2016)

8. Boßelmann, S., Wickert, A., Lamprecht, A-L., Margaria, T.: Modeling directly executable processes for healthcare professionals with xmdd. In: Pfannstiel, M.A., Rasche, C. (eds). Service Business Model Innovation in Healthcare and Hospital Management. Springer Verlag, November 2016. ISBN 978-3-319-46411-4
9. Boßelmann, S., Neubauer, J., Naujokat, S., Steffen, B.: Model-driven design of secure high assurance systems: an introduction to the open platform from the user perspective. In: Margaria, T., Solo, A.M.G. (eds). The 2016 International Conference on Security and Management (SAM 2016), Special Track End-to-End Security and Cybersecurity: From the Hardware to Application, pp. 145–151. CREA Press (2016)
10. Celebic, B., Breu, R., Felderer, M.: Traceability types for mastering change in collaborative software quality management. In: Steffen, B. (ed.) Transactions on FoMaC I. LNCS, vol. 9960, pp. 242–256. Springer, Heidelberg (2016)
11. Doedt, M., Steffen, B.: An evaluation of service integration approaches of business process management systems. In: Proceedings of the 35th Annual IEEE Software Engineering Workshop (SEW 2012) (2012)
12. Fitzgerald, B.: Software crisis 2.0. In: Keynote at EASE 2016, 20th International Conference on Evaluation and Assessment in Software Engineering, Limerick (Irealnd), June 2016
13. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002). doi:10.1007/3-540-45923-5_6
14. Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.-D.: Efficient regression testing of CTI-systems: testing a complex call-center solution. Ann. Rev. Commun. Int. Eng. Consortium (IEC) **55**, 1033–1040 (2001)
15. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: Test Conference, Proceedings, ITC 2003, International, vol. 1, pp. 971–980, October 2003
16. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. Mach. Learn. **96**, 1–34 (2013)
17. Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Burdy, L., Cheon, Y., Poll, E.: An overview of jml tools and applications. STTT **7**(3), 212–232 (2005)
18. Margaria, T., Boßelmann, S., Doedt, M., Floyd, B.D., Steffen, B.: Customer-oriented business process management: visions and obstacles. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 407–429. Springer, London (2012)
19. Margaria, T., Floyd, B.D., Gonzalez Camargo, R., Lamprecht, A.-L., Neubauer, J., Seelaender, M.: Simple management of high assurance data in long-lived interdisciplinary healthcare research: a proposal. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8803, pp. 526–544. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45231-8_44
20. Margaria, T., Raffelt, H., Steffen, B.: Analyzing second-order effects between optimizations for system-level test-based model generation. In: Test Conference, Proceedings. ITC 2005, IEEE International. IEEE Computer Society, November 2005
21. Margaria, T., Steffen, B.: Backtracking-free design planning by automatic synthesis in METAFrame. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 188–204. Springer, Heidelberg (1998). doi:10.1007/BFb0053591
22. Margaria, T., Steffen, B.: Lightweight coarse-grained coordination: a scalable system-level approach. Softw. Tools Technol. Transf. **5**(2–3), 107–123 (2004)

23. Margaria, T., Steffen, B.: Agile IT: thinking in user-centric models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 490–502. Springer, Heidelberg (2008). doi:10.1007/978-3-540-88479-8_35
24. Margaria, T., Steffen, B.: Business process modelling in the jABC: the one-thing-approach. In: Cardoso, J., van der Aalst, W. (eds.) Handbook of Research on Business Process Modeling. IGI Global, Hershey (2009)
25. Margaria, T., Steffen, B.: Service-orientation: conquering complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer, London (2012)
26. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992)
27. Rensink, A.: Model patterns: the quest for the right level of abstraction. In: Steffen, B. (ed.) Transactions on FoMaC I. LNCS, vol. 9960, pp. 47–70. Springer, Heidelberg (2016)
28. Scheer, A.-W.: Architecture of integrated information systems (ARIS). In: DIISM, pp. 85–99 (1993)
29. Steffen, B.: LNCS transaction on the foundations for mastering change: preliminary manifesto. In: Steffen, B., Margaria, T. (eds.) ISoLA 2014. LNCS, vol. 8802, p. 514. Springer, Heidelberg (2014)
30. Steffen, B., Margaria, T.: METAFrame in practice: design of intelligent network services. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 390–415. Springer, Heidelberg (1999). doi:10.1007/3-540-48092-7_17
31. Steffen, B., Margaria, T., Claßen, A., Braun, V.: The METAFrame'95 environment. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 450–453. Springer, Heidelberg (1996). doi:10.1007/3-540-61474-5_100
32. Steffen, B., Margaria, T., Raffelt, H., Niese, O.: Efficient test-based model generation of legacy systems. In: Proceedings of the 9th IEEE International Workshop on High Level Design Validation and Test (HLDVT 2004), pp. 95–100. IEEE Computer Society Press, Sonoma, November 2004
33. Steffen, B., Naujokat, S.: Archimedean points: the essence for mastering of change. In: Steffen, B. (ed.) Transactions on FoMaC I. LNCS, vol. 9960, pp. 24–46. Springer, Heidelberg (2016)
34. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inform. Syst. **30**(4), 245–275 (2005)
35. Beeck, M., Braun, V., Claßen, A., Dannecker, A., Friedrich, C., Koschützki, D., Margaria, T., Schreiber, F., Steffen, B.: Graphs in METAFrame: the unifying power of polymorphism. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 112–129. Springer, Heidelberg (1997). doi:10.1007/BFb0035384
36. White, S.A., Miers, D.: BPMN Modeling and Reference Guide. Future Strategies Inc., Lighthouse Point (2008)

http://www.springer.com/978-3-319-46507-4

Transactions on Foundations for Mastering Change I
Steffen, B. (Ed.)
2016, XII, 257 p. 90 illus., Softcover
ISBN: 978-3-319-46507-4