Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 224 (2023) 91–97

Procedia Computer Science

www.elsevier.com/locate/procedia

The 20th International Conference on Mobile Systems and Pervasive Computing
August 14-16, 2023, Halifax, Nova Scotia, Canada

Residuality and Representation: Toward a Coherent Philosophy of Software Architecture.

Barry M O'Reilly*

Black Tulip Technology
Department of Engineering and Innovation, Open University Milton Keynes UK

Abstract

Software architecture involves making decisions about the structure of applications in respect to the environment in which software must execute. In the process of making these decisions architects produce representations of both the software and the environment. These representations take the form of diagrams, models, textual descriptions, and sometimes loose conversations. Modern approaches to software architecture, such as agile, or Enterprise Architecture approaches, do not attempt to challenge traditional modes of representation. Residuality Theory states that an architecture can be expressed as a stack of residues, or leftovers, that are the results of stressors, unpredictable events that may arise in the environment. This creates a dynamic representation that expresses temporality, flux, and uncertainty as opposed to the traditional static representations of component structure, patterns, and process diagrams that constitute most architectural approaches. Residuality thus entails a challenging conceptual shift and it is necessary to highlight this in order to avoid confusion in practitioners and to make clear just how stark the differences between residuality and traditional approaches are. The aim of this conceptual article is to make it easier for architects to understand the differences and avoid falling back toward default approaches. This article will describe the three important underlying concepts in residuality that inform this shift– processuality, criticality, and difference.

Keywords: Type your keywords here, separated by semicolons ;

* Corresponding author.
E-mail address: Barry@blacktulip.se

10.1016/j.procs.2023.09.015

1. Introduction

The representation of software structures and the environment they execute in is a difficult endeavor. It is challenged by the rate of change in enterprise environments and the ability of the enterprise to articulate and understand this changing environment. Architectural representation thus becomes quickly outdated and remedying this is expensive and difficult. In this dynamic environment any representation is a radical reduction, and much information must be excluded. At the same time, reduction to a representation is vital for any collaborative work to be done at all. This paper proposes that traditional methods of representation are harmful to the practice of software architecture and that residuality theory presents an opportunity to both understand better and improve how software architects think about representation.

2. Problems with Representation in Traditional Architecture

The problem of representation is not unique to software engineering. Representation of substance is seen as a primary focus of western philosophy and is focused on objects and their properties. This approach to understanding the world is known as substance philosophy. In the 2020 paper "The Philosophy of Residuality Theory" [1] a critique of inherent philosophical positions of software architecture was offered. Platonic essentialism, structuralism, machine metaphors of cybernetics, and belief in simple causalities as outlined by Stacey were all seen to contribute to an implicit, unconscious philosophy which directs how architects interact with the world around them. There is, at the heart of this, an unconscious bracketing, limiting the scope of investigation to focus on defining an accessible subset of objects and their properties in detail.

Substance philosophy leads to a world where architectural representation is focused on the categorization and description of static objects, their relations, and their properties. These include components, and even business processes which, despite being processes, are treated as static things. To make this more efficient, the concept of patterns, static collections of static entities, are used to describe collections of these objects and their relations. These representations stand in contradiction to the real world, which is in a state of constant flux, and traditional architectural representations in the form of processes, capabilities, and patterns are vulnerable given their static nature.

The imposition of static representation on an environment in constant flux leads to an inevitable removal of information about the environment, known as model compression, forcing the architect to rely on implicit information and intuition to navigate these environments. Thus, architectural representation is always lacking, missing important information which invites misunderstanding. Software architecture as practiced thus makes it even more difficult to create architecture that is commensurate with the real world. This creates a tension between the static representation and fluctuating reality. The representation will constantly be challenged by unexpected changes and become outdated very quickly. Software that is built upon these representations will thus be brittle and lacking in quality.

It has historically been difficult to see another way to solve this problem. Software components must be represented statically in order to be implemented. Thus, there is a static element of software and an environmental element under constant flux, with each influencing the other. This condition I define as hyperliminality [2], and the architect is required to understand the differences and the resulting interplay between these two conditions. Software architecture has not developed any theory or means of representing this flux. There are many tools to represent static software structures – and unfortunately these approaches are often extended to represent the business environment in which software must execute, unaware of the dangers of model compression in a hyperliminal system.

Attempting to precisely define something which is constantly changing is impossible, and software engineering has fallen into the trap of trying to do this by assuming that the environment was a similar phenomenon to the code.

3. Philosophy and Representation

The problem with the ability to properly represent reality goes back to Plato's cave. David Hume separated the idea of an object in the world from our impression of it, arguing that it was not the same thing. A helpful construct appears in Kant's 'A Critique of Pure Reason' [3] which introduces the concept of noumena and phenomena. For this article, noumena describes something that exists in reality, independent of our senses. This noumena can only be experienced through our senses. This sensing of the noumena is called the phenomena. When something is experienced as simple and ordered, it can be assumed that the noumena is close to the phenomena, or in any case close enough. When they are sufficiently close it is possible to predict and control the properties of the noumena. When this cannot be done, it can be stated that there is a gap between the noumena and the phenomena. I describe this as the phenomenal gap. When this phenomenal gap exists it leads to confusion, surprises, and many things often described as aspects of complexity. The behavior of the noumena with a phenomenal gap mirrors descriptions of complex systems such as as non-linearity (not in line with phenomenal expectations), emergence (has behaviors not included in or predicted by the phenomena) and side effects of intervention such as unintended consequences. There are two aspects to this phenomenal complexity, firstly when a phenomenal gap is due to ignorance, and secondly when the environment in question is in such a state of flux that it cannot be adequately represented. For the remainder of this article only the latter is considered.

The phenomenal gap for software itself is obviously small. It is possible to test, predict, change and control the behavior of software, evidenced by the trillions of lines code currently running across the world. There is, however, an enormous phenomenal gap in our understanding of any enterprise environment, or indeed any socio-economic system. As software must exist in the environment, and its structure is defined by our representation of this environment, this creates stress on any software architecture based on these representations.

If this is really a problem, why is it not discussed more and why has the problem of representation persisted for so long without redress? Kuhn [4] described the fall of a paradigm into crisis when anomalies reach a certain tipping point, and at that point a discipline would struggle to update its existing knowledge to explain anomalies before eventually giving way to a new paradigm. The world of software engineering seems stuck in Kuhnian crisis in the absence of a new paradigm. In response to this crisis, the agile movement arose. However, the agile movement focused on the process of developing software and retained the static representations unquestioningly, thus ensuring that the problem remained and was left to a flexible development process to fix.

The inherent and unquestioning belief in substance philosophy leads to the impulse to hurriedly identify objects and their properties and categorize these. The combination of this and the phenomenal gap that exists in describing an enterprise software environment leads to a constant problem for software architects – dependency on a representation that is destined to be incorrect. It can, therefore, be argued that correcting this problem will require a change in our philosophical approach to representation.

4. Residuality

Residuality is a process philosophy. That is, residuality sees dynamic becoming as the defining structure of the world, as opposed to the substance view of interrelated entities. Residuality describes a world of constant change. It is not speculative in that it does not try to account for the origins or reasons for the processes, nor to categorize or analyze these processes, but simply views the things we see around us as processes. Residuality is a skeptical, Humean treatment of complexity theory - reducing complexity to the minimum of what can be said, removing, as far as possible, opinion and speculation.

This does not mean, however, that substance is ignored. Unlike other process philosophies, residuality sees substance as residue, accidental leftovers of indeterminate, often invisible and indescribable processes. The presence of a phenomenal gap usually indicates that such processes are at work.

Residuality provides an interface to the world of both substance and process through the residue. The residue is a loosely defined structure that describes the differences in a software application's structure under different environmental conditions [2].

To bridge the gap between the processual view and the substance view, residuality uses the work of Kauffman [5] to link them. In Kauffman's NK model, interactions between substances move systems from one state to another, and this number of states is limited by the degrees of freedom that elements make possible for each other in their interactions. These limited number of states are known as attractors. These attractors form a network and are a structure in the unstructured – a substance view of process. The interactions between elements are seen as driving the process, and the process, in turn, affects the elements, creating a series of residues over time. In this sense, complexity, or a complex system, is not a 'thing' to be categorized but a self-sustaining process and this infers that complexity is also causation. The only thing needed for interaction and movement is proximity. Thus, substance and process are essentially the same thing, creating and driving each other. The dynamic processes unleashed cannot be measured, mapped, or controlled, given that the substances that drive them are constantly changed by the processes themselves. This is similar to Ralph Stacey's 'transformative' causality [6] and Alicia Juarerro's work on constraints as causality and self-cause [7]. In this case, it is impossible to engage with either the changing substance or the unknowable process with traditional methods, and so the residue becomes the interface with which the architect can interact with the attractors in the system through the components in the software. This has the advantage that the architect can work without ignoring either process or substance perspectives. However, this is with the caveat that an architect must now work with many residues and their transitions, not just a simple static representation. This means that traditional methods of representation will not be enough to capture the architect's newfound understanding of the architecture. This also requires a rethinking of traditional architectural processes as described in "Residuality Theory, random simulation and attractor networks" [2].

4.1. Criticality

When a system is able to survive across different attractors, it is said to have criticality – an internal structure capable of reorganizing to survive in other attractors. Reaching this criticality is the goal of residuality theory, which is a different goal than that of 'correctness' – a hangover from the mathematical and computer science roots of software engineering. How this is achieved and measured is detailed in the 2022 paper "Residuality theory, random simulation and attractor networks" [2].

The modeling of residue allows architects to engage with the system at a substance level whilst ensuring that the environment and the application are always viewed as processual, without immediately using static representations for either. It also shifts the goal of the software application from achieving correctness to achieving criticality.

5. Representing Through Difference

Once this combination of residuality, hyperliminality and the attractors of the NK model have been resolved to answer the question about how software systems in dynamic environments should be considered, it is time to look at the third important concept in residuality – which is difference. It is important to realize that individual residues are not models of the entire environment or software structure, instead they involve only those pieces affected by the move to a particular attractor. Some elements of an environment will have a small phenomenal gap and it is unnecessary to constantly repeat representation of these in every residue. Thus, residues consist only of that which is different between each residue. These differences are represented as text descriptions in the stressor analysis, and as rows in incidence matrices [2]. Static representations of component structures are arrived at through these fluctuating representations, which each fluctuation potentially giving rise to new static representations. It is not a new idea to

focus on difference. The work of Bateson[8], and the philosophies of Deleuze and Derrida [9] can be considered philosophies of difference.

Derrida's concept of deconstruction has interesting parallels to residuality and his work is linked to the complexity sciences by Cilliers [10]. Deconstruction is the challenging of incumbent structure and the seeking of fault lines and is relevant in the inevitable need for residuality to break down the stubborn, projected substance structures in the phenomenal gap, including traditional architecture tools.

Inspired by Bergsonian processuality and the concept of difference, it is Deleuze who brings a metaphysical investigation that foreshadows residuality in the closest way. Deleuze rejects the Platonic, substance-fueled tendency to reduce things to their substance-based identity, instead emphasizing the importance of change, the unending process of differential change through repetition and the defining of identity through these differences [11]. This mirrors the stressor analysis in residuality [2], provoking change in a repetitive pattern and capturing the differences in the form of residues. The Deleuzian dialectic described by Williams [12] has many similarities with the stressor analysis described in [2] - the random generation of events in the environment designed to identify residues. This is influenced by Spinoza's claim that a thing is more about its conditions than its properties, a statement just as relevant to residuality as it is to Deleuze's philosophy.

The Deleuzian idea of difference and repetition is best explained through the metaphor of a walk. The first time we take a walk, we experience just one view of the walk. As we take the same walk over time, we notice differences. The walk for us is defined not by that first walk, but the sum of the differences between all the subsequent walks across seasons, years, and changes in paths, buildings, and the surrounding environment. The phenomenon is thus created repeatedly and is much richer for its exposure to differences. This repetition with focus on difference is essential to Deleuze's way of seeing the world. The model implied by residuality mirrors this worldview quite well. Residuality integrates these Deleuzian instincts into an empirical framework that allows both exploration and the embrace of process philosophy, and at the same time the production of representations and proof of efficiency [2]. This concept of a walk can also be used to explain the model of reflective practice, relevant for architects, in Donald Schön's "The Reflective Practitioner" [13].

In the iterations of agile development, or the assessing from different viewpoints of Enterprise Architecture approaches, repetition with focus on difference can be seen to be instinctively, although implicitly, present in most modern approaches to software engineering. In standard approaches to architecture, the architect engages in the production of many different models. These include variations on requirements engineering, risk analysis, design, design testing and estimation, among many others. The execution of these methods is, in modern times, not seen to deliver anything particularly accurate in terms of describing the environment. In fact, the agile movement has made the industry deeply suspicious of the ability to gather complete information before implementation work begins. Given that these methods do not deliver what they actually intend, it can be seen that each method is actually a metaphorical walk, engaging with the wider system with a slightly different focus each time, noticing differences and tensions, and updating the overall understanding of the architecture and the environment. This offers an explanation for why these methods often contribute to successful projects, without actually delivering accurate requirements, risks, or estimations. The generation of each model is a Deleuzian walk, and the architects understanding of the hyperliminal environment, including the software and the wider enterprise, grows with each walk. By focusing on difference and rejecting the substance-based worldview, residuality simply forces the architect to take more walks, and dispenses with the theatre of producing substance focused static models as byproducts of the process.

There is danger in introducing the work of postmodern philosophers to engineering work. Post-structuralism has been ridiculed as "fashionable nonsense" and is criticized for adding no new knowledge. However, here residuality is merely drawing parallels with post-structuralist thinking and noting the link between this and complexity as described by Cilliers. In fact, the complexity sciences, post-structuralism, Schön's contrasting of reflective practice and technical rationality are all aspects of the phenomenal gap, describing the same problem from different vantage points. Perhaps the biggest critic of post-structuralism can be found in the analytic tradition, but even late Wittgenstein's mellowing

toward the blurry picture can be seen as an admission that different tools are necessary in the phenomenal gap. Bergson's claim that science did not yet have a metaphysics echoes this continuing struggle, one which is all too apparent in the stumbling philosophy behind software architecture. The unrelenting focus on substance and static representation leads to Whitehead's "fallacy of misplaced concreteness" [14] in the tools of traditional architecture and the deconstructive methodologies of post-structuralism are actually helpful in moving past this tendency.

The conclusion here is that the post-structural ideas of Deleuze and Derrida were entirely reasonable conjectures in the presence of the phenomenal gap and substance metaphysics, but that Platonic demands for generalization made them appear vague and difficult to apply to real problems. Residuality, thus, forces a sharpening of these ideas and brings them to an arena that allows their use in scientific, empirical investigation.

6. Representation of residue.

Residuality allows for the representation of differences in flux by use of simple textual descriptions combined with matrices that change as understanding changes [2]. These can also use visual descriptions if wished – in fact the actual mode of representation is left to the individual architect. The important thing is not the actual representation but clearness about what should be represented and what that representation actually means: instead of representing static entities, residuality represents differences and flux. This cannot be precise, by definition, but requires that we ask questions of our current understanding of the environment. To represent the entire residual stack is never possible, but small glimpses of it are enough to trigger the journey toward criticality. Creating each element in these fuzzy representations encourages further "walks". Comparing the representation of residues directly with traditional methods or architectural representation, such as UML, would be to miss the point and reinstate the substance paradigm. Traditional static representations of software structures are instead produced through the representations of the residues and are the end result of the process of contagion analysis [2]. These static representations are never changed in themselves, instead generated anew through the process of residual analysis as things change in our understanding of the environment.

While traditional engineering techniques have been appropriated within software engineering, their identity focused approach to representation is made easier by the fact that there is a smaller phenomenal gap in physical engineering. The application of these ideas to software engineering, and the failure to arrive at the right results led a doubling down, focusing first on language, then on machine metaphors to control the environment, and eventually to agile, which unfortunately avoided the issue by keeping identity focused representation. Engineering approaches have tried to remove, simplify, reduce, or encapsulate complexity, in order to preserve the concept of identity and the centering of the object. Even systems thinking approaches and Complex Adaptive Systems (CAS) focus on the identity, the system, and its properties and behaviors, and even resilience is commonly defined as a return to identity.

In traditional architectural representations, the architect has had to carry the representation of difference, processuality, and criticality internally, as an extension of the static representations incapable of describing these things. In traditional architecture, representations are often in tension with each other because of this lacking, and serious architectural work involves not producing the static representation, which is trivial, but silently investigating the tensions caused by substance representations, with no way of representing this critical work. Residuality allows this difference, processuality, and criticality to be represented. However, it is also necessary to note that, by definition, a residue will never adequately represent reality. The job of the representation is to capture differences and trigger "walks". It is important that architects are aware that there is no representation of residue that is correct. This can be an incredibly difficult concept to grasp – but residues do not need to be accurately represented for the resulting architecture to demonstrably reach criticality.

7. Conclusion

Residuality embraces the use of post-structural thinking in order to manage the phenomenal gap but remains pragmatic and focused on empirical verification. It introduces the concepts of process, difference, and criticality as the foundation of a new philosophical approach to software architecture that is very different than substance-centered

approaches. This philosophical position allows architects to see more of the world that they are engaging with and provides explanations for the flux and temporality that all architects know is present in their work but has been made a second-class citizen due to focus on static representation. This is an enormous change in how architects view the world, and challenges the current paradigm of which most architects are not even aware they are operating in.

The residue is a centering of our fallibility - the idea that things will crumble, and that stress, flux, and volatility cannot be engineered or wished away. It is a concept that represents codified learning, reflective practice, and is open to empirical inquiry. It owes its roots to the complexity sciences and post-structural thinking but is not strictly faithful to either or any one interpretation of these. Residuality is better described as anti-structuralism – constantly warning of the risks of structuralism and providing a means to question and mitigate the risk of naïve structure. Residuality is used to manage the fallout of the failure of structural models of the environment, a constant consequence of complexity. In residuality, structure in the environment is fleeting, and structure in software is significant. Therefore, software structure cannot be based on any imagined structure in the environment, but must correlate instead to the attractors between which any structure fluctuates. It is a coherent, novel, and pragmatic philosophy of software architecture that fills the vacuum in traditional approaches.

This article has aimed to highlight that residuality theory is not just another idea within the traditional software engineering paradigm, but that it involves an underlying shift in philosophical position that must be made clear to fully grasp the concept. Representation in residuality is not commensurate with traditional architectural representation, and the two should not be directly compared. Instead, the use of residuality to produce traditional static component representations should be seen as the bridge between these two worlds.

References

[1]O'Reilly, B. M. (2021). The Philosophy of Residuality Theory. Procedia Computer Science, 184, 809-816.

[2]O'Reilly, B. M. (2022). Residuality Theory, random simulation, and attractor networks. Procedia Computer Science, 201, 639-645.

[3]Kant, I. (1908). Critique of Pure Reason. 1781. Modern Classical Philosophers, Cambridge, MA: Houghton Mifflin, 370-456.

[4]Kuhn, T. S. (2012). The structure of scientific revolutions. University of Chicago press.

[5]Kauffman, S., & Kauffman, S. A. (1995). At home in the universe: The search for laws of self-organization and complexity. Oxford University Press, USA.

[6]Stacey, R. D. (2009). Complexity and organizational reality: Uncertainty and the need to rethink management after the collapse of investment capitalism. Routledge.

[7]Juarrero, A. (2000). Dynamics in action: Intentional behavior as a complex system. Emergence, 2(2), 24-57.

[8]Bateson, G. (1970). Form, substance and difference. Essential readings in biosemiotics, 501.

[9]Cisney, V. W. (2018). Deleuze and Derrida: Difference and the Power of the Negative. Edinburgh University Press.

[10]Cilliers, P. (2002). Complexity and postmodernism: Understanding complex systems. routledge.

[11]Deleuze, G. (1994). Difference and repetition. Columbia University Press.

[12]Williams, J. (2013). Gilles Deleuze's Difference and repetition. Edinburgh University Press.

[13]Schon, D. A. (1968). The reflective practitioner. New York.

[14]Whitehead, A. N. (1925). Science and the modern world: Lowell lectures, 1925. New American Library.