On the Operationalization of Composable Architecture by Means of Normalized Systems Theory

Geert Haerens
Antwerp Management School, Belgium
Engie nv, Belgium
Email: geert.haerens@engie.com

Herwig Mannaert
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerpen.be

***Abstract*—In a fast-evolving world, companies require IT solutions that allow them to adapt swiftly to changing conditions. In 2020, Gartner introduced the Composable Architecture Framework as a guiding principle to create application landscapes that are easily composable and re-composable, thus supporting change. The Normalized Systems theory is about the creation of evolvable modular software. The concepts presented in the Composable Architecture Framework resonate with Normalized Systems. A closer analysis of the framework through Normalized Systems theory reveals that Gartner's framework lacks precision. As such, following the guidance of the framework will insufficiently protect companies from change, both outside and inside the organization.**

Keywords*—*Normalized Systems Theory; Composable Architecture; Packaged Business Capabilities.

I. Introduction

For many years now, a death wish toward the monolithic application has been declared. Monolithic applications are difficult to change and unsuitable in our fast-moving world. Many paradigms have been proposed over the years that aim at splitting applications into smaller, modular parts. With the rise of the Internet and faster network speeds, the physical distribution of those parts has become a reality. Distributed Computing Environment (DCE) [1] proposed modules encapsulating functionality that could be activated using Remote Procedure Calls (RPC). An essential benefit to splitting applications into smaller, independent callable modules is reuse. This resonates with McIlroy's dream, expressed during the 1968 NATO conference on Sofware Engineering, where *" ... I expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able to safely regard components as black boxes."*. SAP ERP (Enterprise Resource Planning), for instance, uses this kind of approach to allow the calling of SAP functionalities from other systems through Remote Function Calls (RFC). SAP re-baptized RFC to BAPI, Business Application Programming Interface, to focus even more on encapsulated functionality. In the past couple of years, we have seen a shift from encapsulated functionality toward technology, meaning that today, the talk of the town is about REST APIs, message queues, and event systems as a means of implementing integration between modules but without paying attention to the functionality.

In 2020, Gartner [2] introduced the notion of Packaged Business Capabilities (PBC) to create Composable Applications. They proposed the Composable Architecture Framework [2] as a guide for proper encapsulating functionalities and using technologies that allow flexibility in recomposition and evolution. Their approach connects the previous focus on functionality with today's emphasis on technology.

Normalized System theory (NS) [3], originating in software development, put forward the necessary conditions for the evolvability of modular structures. In this paper, we will try to operationalize and guide the implementation of the concepts of Gartner's Composable Architecture Framework by using NS [4]. The paper is structured as follows: In Section II, we will introduce Gartner's Reference Architecture for Composable Business Technology, followed by Section III that will introduce NS. In Section IV, we refer to related work, and in Section V, we attempt to operationalize the framework through NS. Section VI reflects on the operationalization and the paper is wrapped up in Section VII.

II. Gartner's Reference Architecture for Composable Business Technology

In 2020, the world was struck by COVID. Besides the human loss and suffering, businesses were also severely disrupted by this crisis. Gartner noticed that companies with a modular application approach could adjust swiftly to external conditions by quickly and safely assembling, disassembling and reassembling applications as the world required. In November 2020, Gartner published their Reference Architecture for Composable Business Technology [2], which investigates the conditions required for a platform to facilitate the composition and re-composition of applications. The needed ingredients for such a platform are PBCs as the essential modules, an application composition experience allowing custom assembly of PBCs via low code and no code, an application composition platform enabling development and deployment of the newly composed applications, and a data fabric allowing easy access to data and analytics on those data.

The PBCs are modules that encapsulate a well-defined business capability (recognized by the intended business users) and must adhere to the following conditions:

- They are modular and cohesive.
- They are autonomous (can run independently) and have minimal dependencies with external components.
- They allow orchestration as they can realize a process flow across PBCs (via APIs, events, etc.).

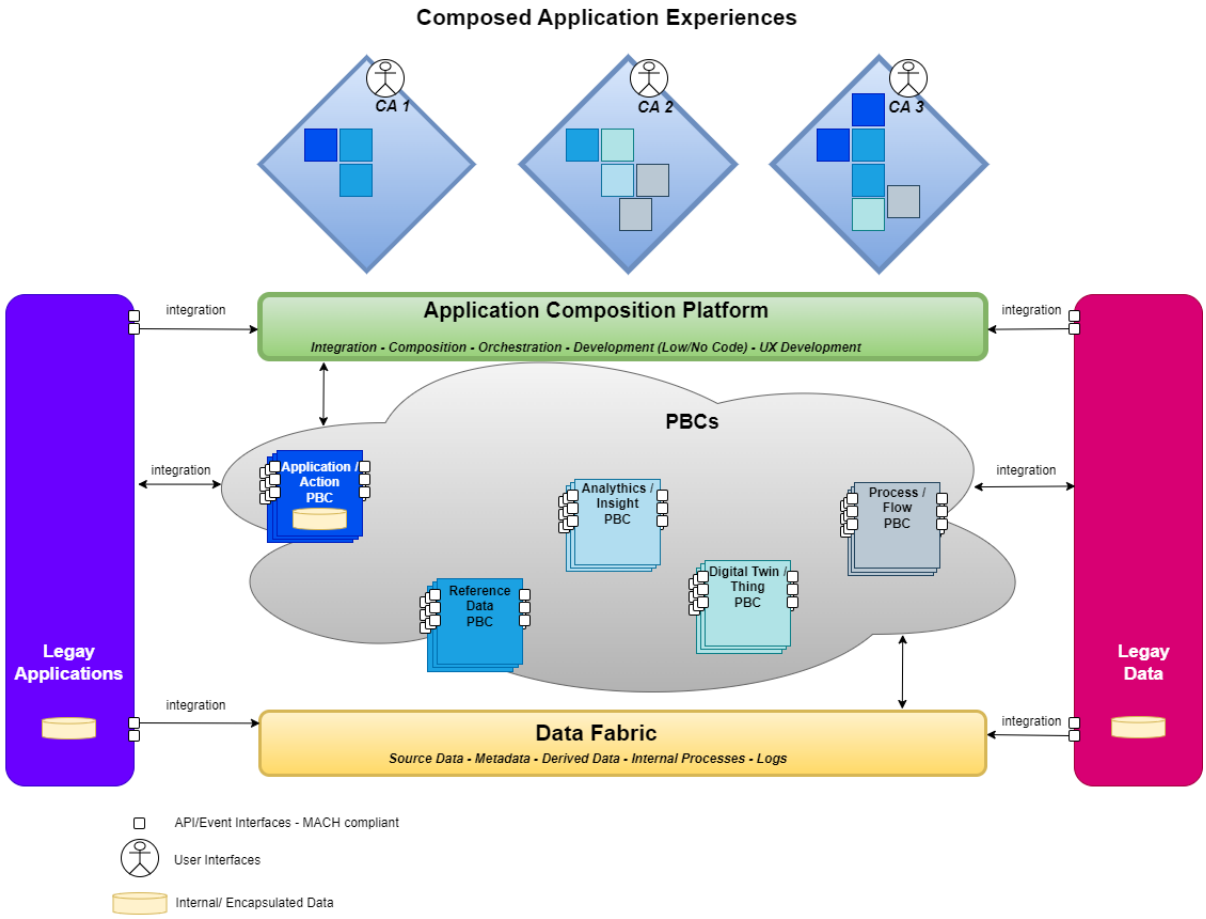

Fig. 1. Overview of the Composable Architecture Framework [5].

- They are discoverable as easily accessible and recognizable by those who require them.

Defining the right level of business capability granularity is challenging (too large = monolith, too small = more complex to identify).

Gartner provides further guidance on the PBC definition by defining PBC types.

There is the Application Type PBC that encapsulates both data and functions related to a well-defined business capability. Application Type PBC can be used to create fully expressed (=autonomous and encapsulating a full context) PBCs or basis business function PBC (= not autonomous and encapsulating a part of a context). Application Type PBC can create pseudo PBCs that are APIs toward existing monolithic applications. They encapsulate the legacy application, allowing them to participate in PBC composition but lack some flexibility in true PBCs regarding their evolvability.

Reference-type PBCs allow encapsulated access to data in the data fabric. They can access master data, metadata, or any data instance representing a business object or physical data container.

The Insight Type PBC allows the encapsulation of data analytics processes. It can perform analytic operations or apply AI models (ML, Deep learning, etc.) to data in the data fabric.

The Thing Type PBC encapsulates things in the physical world. It can be used to access and manipulate data from the real world (IoT).

There is the Flow Type PBC that combines different PBC in an order. Flow Type PBCs facilitate the creation of orchestrated PBCs that encapsulate a process.

The PBCs are activated via event channels and called via APIs. They can also provide different and optional user interfaces (web, mobile, or other).

In Table 1, Gartner contrasts the organization of an application landscape in PBCs (combined with a composition and deployment platform) with traditional application landscapes.

Garter analyzed the main change drivers in their papers: adding new business capabilities and their associated PBCs. Gartner further emphasizes that a platform that conforms to the above specifications is insufficient. Business and IT must work closely together to define and implement the required business capabilities, requiring what Gartner calls fusion teams. These teams are essential in creating business–IT alignment and, thus, value creation.

To start with a composable architecture strategy, one must first know what PBCs are already in the company. They may

TABLE I
CONTRASTING TRADITIONAL WITH PBC APPLICATION LANDSCAPES (FROM [2])

Criteria	Traditional Applications	PBCs
Primary value	Business capability	Business capability
Primary access	User Interface (UI)	Programmatic Interface (API, event)
Scope	Many business objects	One business object
Internal architecture	monolith or modular	monolith or modular
Designed for	Business	Business and IT
Design priority	Stability	Agility
Delivered value	Business solutions	Recomposable business solutions
Production style	Project	Product
Essential tools	Customization included	Composition, added cost
Required IT skills	Customization, low	Composition, high
Cost	Bulk, some ”shelfware”	Componentized, tracks value
Governance	Sample	Complex
Internal data	”Owned”	”Owned”
Open for integration/composition	Partially, a secondary priority	Fully, primary design objective

already be present as fine-grained PBCs or via applications aggregating PBCs and accessible via APIs. It is vital that investments in future technologies that should provide PBCs can be used as such. The individual PBCs must be accessible via APIs and not in an aggregated way. For example, a SaaS solution integrating multiple PBCs should allow the individual usage of the platform PBCs without depending on the other PBCs.

New applications are composed of assembled PBCs, allowing faster delivery and updating. Updating a PBC in the catalogue should update all applications with that PBC as an active module.

In summary, the Gartner Composable Architecture Framework *“presents the reference model for developing business applications that are modular, composable, easily adapted and ready for change.”* [2].

III. FUNDAMENTALS OF NS THEORY

Software should be able to evolve as business requirements change over time. In NS theory [6], the lack of Combinatorial Effects measures evolvability. When the impact of a change depends not only on the type of the change but also on the size of the system it affects, we talk about a Combinatorial Effect. The NS theory assumes that software undergoes unlimited changes over time, so Combinatorial Effects harm software evolvability. Indeed, modifications to a system depend on the size of the growing system. In that case, these changes become more challenging to handle (i.e., requiring more work and lowering the system’s evolvability).

NS theory is built on classic system engineering and statistical entropy principles. In classic system engineering, a system is stable if it has bounded input, which leads to bounded output (BIBO). NS theory applies this idea to software design, as a limited change in functionality should cause a limited change in the software. In classic system engineering, stability is measured at infinity. NS theory considers infinitely large systems that will go through infinitely many changes. A system is stable for NS if it does not have Combinatorial Effects, meaning that the effect of change only depends on the kind of change and not on the system size.

NS theory suggests four theorems and five extendable elements as the basis for creating evolvable software through pattern expansion of the elements. The theorems are proven formally, giving a set of required conditions to follow strictly to avoid Combinatorial Effects. The NS theorems have been applied in NS elements. These elements offer a set of predefined higher-level structures, patterns, or “building blocks” that provide a clear blueprint for implementing the core functionalities of realistic information systems, following the four theorems.

A. NS Theorems

NS theory [6] is based on four theorems that dictate the necessary conditions for software to be free of Combinatorial Effects.

- Separation of Concerns
- Data Version Transparency
- Action Version Transparency
- Separation of States

Violating any of these four theorems will lead to Combinatorial Effects and, thus, non-evolvable software under change.

B. NS Elements

Consistently adhering to the four NS theorems is challenging for developers. First, following the NS theorems leads to a fine-grained software structure, which introduces some development overhead and may slow the development process. Second, the rules must be followed constantly and robotically, as a violation will introduce Combinatorial Effects. Humans are not well suited for this kind of work. Third, the accidental introduction of Combinatorial Effects results in an exponential increase in rework.

Five expandable elements [7] [8] were proposed, which make the realization of NS applications more feasible. These elements are carefully engineered patterns that comply with the four NS theorems and that can be used as essential building blocks for various applications: data element, action element, workflow element, connector element, and trigger element.

- **Data Element**: the structured composition of software constructs to encapsulate a data construct into an isolated

module (including get- and set methods, persistency, exhibiting version transparency, etc.).

- **Action Elements**: the structured composition of software constructs to encapsulate an action construct into an isolated module.
- **Workflow Element**: the structured composition of software constructs describing the sequence in which action elements should be performed to fulfil a flow into an isolated module.
- **Connector Element**: the structured composition of software constructs into an isolated module, allowing external systems to interact with the NS system without calling components statelessly.
- **Trigger Element**: the structured composition of software constructs into an isolated module that controls the system states and checks whether any action element should be triggered accordingly.

The element provides core functionalities (data, actions, etc.) and addresses the Cross-Cutting Concerns that each of these core functionalities requires to function correctly. Cross-cutting concerns cut through every element, requiring careful implementation to avoid introducing Combinatorial Effects.

C. Element Expansion

An application comprises data, action, workflow, connector, and trigger elements that define its requirements. The NS expander is a technology that will generate code instances of high-level patterns for the specific application. The expanded code will provide generic functionalities specified in the application definition and will be a fine-grained modular structure that follows the NS theorems (see Figure 2).

The application's business logic is now manually programmed inside the expanded modules at pre-defined locations. The result is an application that implements a certain required business logic and has a fine-grained modular structure. As the code's generated structure is NS compliant, we know that the code is evolvable for all anticipated change drivers corresponding to the underlying NS elements. The only location where Combinatorial Effects can be introduced is in the customized code.

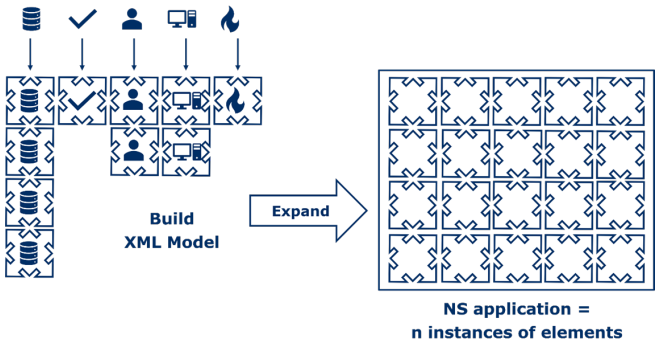

Fig. 2. Requirements expressed in an XML description file, used as input for element expansion.

D. Harvesting and Software Rejuvenation

The expanded code has some pre-defined places where changes can be made. To keep these changes from being lost when the application is expanded again, the expander can gather them and re-inject them when re-expanded. Gathering and putting back the changes is called harvesting and injection.

The application can be re-expanded for various reasons. For example, the code templates of the elements can be improved (fix bugs, make faster, etc.), new Cross-Cutting Concerns (add a new logging feature) can be included, or a technology change (use a new persistence framework) can be supported.

Software rejuvenation aims to routinely carry out the harvesting and injection process to ensure that the constant enhancements to the element code templates are incorporated into the application.

Code expansion produces more than 80% of the code of the application. The expanded code can be called boiler-plate-code, but it is more complex than what is usually meant by that term because it deals with Cross-Cutting Concerns. Manually producing this code takes a lot of time. Using NS expansion, this time can now be spent on constantly improving the code templates, developing new templates that make the elements compatible with the latest technologies, and meticulously coding the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. All developers can use a modification on a code template by one developer on all their applications with minimal impact, thanks to the rejuvenation process (see Figure 3).

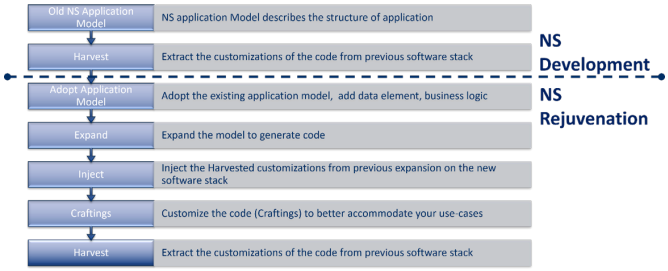

Fig. 3. NS development and rejuvenation.

IV. Related Work

While searching for relevant literature, we found some papers discussing Composable Architecture. The most relevant publications are, on the one hand, a paper about a possible methodology and demonstration by use case for implementing Composable Architecture, and on the other hand, the book of AW Sheers, "The Composable Enterprise" [9]. While Scheer [9] tried to rearchitect a complete organization, Ivas [10] provides a methodology to introduce Composable Architecture and demonstrates it with a use case. The paper also provides a good literature review where the author notices that only a few academic papers discuss Composable architecture (search on

Google Scholar, Web of Science, and Resource Gate). At the same time, thousands of papers are found in the professional literature (via Google), pointing out the need for academic attention to the subject. The paper proposes a methodology consisting of 6 steps (from [10]:

- Understand business drivers and objectives. The first step is to understand the business background of the initiatives, i.e., the rationale, objective, and scope from the business point of view.
- Understand the holistic scope of the initiative. The second step is about understanding which value stream steps and business capabilities from the Holistic value delivery are being affected by the initiative and how.
- Understand the current situation. Understand and sketch the scope of the current solution (architecture).
- Understand the situation and needs at the enterprise level. Identify if any components can be reused or optimised by this solution at the enterprise level or if there are other future initiatives with the same need.
- Design as API -first Headless PBC (preferably according to MACH [11]). If you need to implement a new service, you should preferably design it according to MACH principles. Otherwise, deliver business change by creating new or optimising existing monolith modules by API-first and Headless MACH principles.
- Implement business-IT aligned PBC solution and consolidate. Implement the agreed business-IT solution and consolidate any old solutions into the new solution that implements the same functionality (business capability).

For an Enterprise Architect, those steps are logical and provide excellent guidance. We argue that, next to the need for academic attention to applying Composable Architecture for (re)introducing functional re-uses, there is a need as well for academic attention to properly operationalizing/implementing Composable Architecture to make the dream of McIllroy a reality and not a nightmare.

V. Operationalization of Gartner's Reference Architecture for Composable Business Technology

In this section, we will take the components of the Composable Architecture Framework (see Figure 1) and try to guide how to operationalize them. We start by looking closer at business capabilities and how they should help align business and IT. We continue by looking at the modularity of PBCs and the different types of PBCs defined by the Composable Architecture Framework. We end this section by looking at PBCs and Cross-Cutting Concerns (CCC) and by taking a closer look at the requirements for a PBC platform.

A. Defining Business Capabilities

The concept of business capabilities originates in the resource-based view [12] of companies, where it is considered vital to identify resources and capabilities that provide a competitive advantage [13]. This evolved into the idea that companies need to know "what" kind of activities they are undertaking and gave rise to using the term Business Capabilities. Although decades have passed since the first mention of capabilities, there is no agreed-upon definition of business capabilities and how they should be defined, named and properly used. We refer to [13] for a comprehensive literature overview.

The cornerstone of Gartner's Composable Architecture Framework is PBCs. However, as argued above, the definition of business capabilities can be problematic. If the definitions are unclear, then the implementation into PBC is suspect. For some industries, there are business capabilities frameworks such as BIAN [14] for the banking industry or NBility for energy transmission and distribution in the Netherlands [15]. Still, a lot of sectors have no such frameworks openly available.

As such, a necessary condition for using the Composable Architecture Framework, being well-defined business capabilities, is already a tough nut to crack, and scientific guidance on how to do it is lacking.

B. Business IT Alignment using PBCs

Gartner considered the business capabilities to be well-defined, shared between business and IT, and as an accurate alignment tool between the two. They refer to "Fusion Teams" as the secret formula to create this shared understanding. Fusion teams are teams in which business and IT people are present. Close collaboration between the two will result in shared understanding and better solutions. This idea is also present in Agile Frameworks such as SAFe [16], where agile teams are considered multi-functional (mix of business and IT), and having people who know the job best close together, their emerging designs will be better than those imposed by intentional architecture. Or stated otherwise, the PBCs are to be built bottom-up and not defined top-down.

From NS, we know the importance of having an anthropomorphic design. This means that the naming of modules should represent something that exists in the real world, as this increases understanding of the module's function. The same holds for business capabilities. If they are insufficiently fine-grained and abstract, they no longer represent business reality, but if they are too detailed, the number of business capabilities is considered too large and thus complex. Whether bottom-up or top-down is the right approach is beyond the scope of this paper. One often notices a combination of both; they meet in the middle somewhere.

C. PBCs and Modularity

Gartner's PBCs are modular, where modularity is defined as "partitioned" into a cohesive set of components [2]. Garter further adds that *"The granularity of PBCs, as with all modular systems, is a common design challenge. Modular components that are too large may be easier to manage, but they are harder to change are use in new compositions. Components that are too small may be easier to assemble but harder to isolate, identify, find or change."* [2]. This statement is vague. What are the objective criteria for too large and

too small? Secondly, as neither is considered a good modular design, what are the requirements for a sound module size?

Normalized Systems theory explicitly defines the optimal module size to facilitate anticipated changes. A module must be split into smaller modules until each complies with the four NS theorems: SoC, AvT, DvT, and SoS. When all the theorems are met, the optimal size is reached. As these are the necessary conditions for system stability under change, violating them will introduce Combinatorial Effects (CE).

Separation of Concern (SoC) teaches that each PBC should encapsulate a separate change driver. PBCs must be defined as fine-grained and very specific. For example, a possible PBC is Asset Monitoring. The asset could be an IT system (servers, databases) or an OT system (Operational Technology as SCADAs, Turbines, etc.). Although both are assets, secure monitoring for both requires different implementations. This would mean that if we were only to use one PBC for this, we would need two different versions of this PBC: one for IT and one for OT. It suffices to extend this way of thinking and conclude that insufficiently fine-grained PBCs would lead to multiple versions of the PBC and break the anthropomorphic relation between what something is (a PBC) and how something is done. Such an anthropomorphic relation is a required condition for a PBC platform as PBCs must be easily discoverable (see Section II)

Action version Transparency (AvT) enforces interface stability when the implementation changes. The Composable Architecture Framework does not explicitly mention interface stability when the PBC implementation changes. However, it does say the need for technologies that will result in loosely coupled systems, such as providing interfaces via APIs and event buses and using technologies promoted by the MACH Alliance [11]. We will return to this point when discussing the PBC platform. However, AvT should not be limited to technical protocols and implementations. PBC definitions should also pay attention to semantic issues regarding version transparency. In case a business capability introduces a new feature, a default behavior needs to be defined in case the new feature is not specified.

Data version Transparency (DvT) ensures that evolution in data attributes does not impact processing functions that do not use these new attributes. The Composable Architecture Framework does not mention this concept; it is left to the data fabric. This paper will not discuss the vagueness or evolvability of concepts such as data fabric.

Separation of State (SoS) required all modules to keep state and make their state externally visible. The Composable Architecture Framework does not mention the need for statekeeping of the PBCs. Like with AvT, there is the mention of MACH technologies that promote using process Choreography over process orchestration for evolvability purposes. Independently of whether this is valid, choreography (as orchestration) requires state-keeping and exposition if you want to trace process issues back to the failure location. Similar to AvT, statekeeping should also be looked at semantically, i.e., what is the actual business state at specific points in the process flows.

D. PBC Types

Garter introduces PBC Types, such as application PBC, data entry, analytics PBC, Digital Twin and Process. What is missing is the relation between a PBC and a PBC Type. Is a PBC comprised of multiple PBC types, or do they have a one-to-one relationship? The latter would be strange as to do something, a ”what” or a capability, you require things to make it happen, ”hows”, and the PBC Types are pointing more into the direction of a ”how” than in the direction of a ”what”. The PBC Types have similarities with NS elements.

- Application PBC Type (action) versus the Task Element
- Data Entity PBC Type (reference) versus Data Element
- Process PBC Type (flow) vs Flow Element

For NS, an Analytics PBC would be an Application PBC, where the action/task is to perform some analytic operation. A similar reasoning applies to the Digital Twin (thing) PBC, where this would combine data and task elements in NS. We notice Garter’s difficulty in addressing concepts at their suitable granularity and abstraction level.

E. PBCs and Cross-Cutting Concerns

Normalized Systems recognize that cross-cutting concerns must be treated as any other change driver and require splitting into different modules and proper encapsulation. PBC focus on business, not technology. The technologies used to implement, run, and deploy the PBC are abstracted, and the only type of guidance regarding technology is to use MARCH-compliant technologies. As stated in the previous subsection, we notice an analogy between PBC Types and NS Elements. NS Elements are there to allow proper encapsulation and separation of cross-cutting concerns. One would expect something similar in PBCs, but this is not the case. This represents a lack of design criteria for the PBC and could result in proper SoC regarding capabilities/functionalities but zero evolvability for the cross-cutting concern and associated technologies. As new technological implementations change faster and faster, ignoring this change driver will introduce CEs. Gartner implies leaving this up to the PBC Platform to take care of.

F. PBC Platform

Much of the PBC implementation magic is left over to the PBC Platform. It must facilitate the discovery of the available PBC, the composition of PBCs and the deployment of PBCs. We already argued that proper discovery requires anthropomorphism of the PBC namespace and associated granularity to avoid PBC versioning without meaning. In the previous subsection, we argued that Gartner seems to push the treatment of the cross-cutting concerns toward the PBC Platform. Instead of addressing cross-cutting concerns at the PBC level, it would need to be done at the PBC Platform level. This would be a possibility, were it not that as a selection criteria for a PBC platform, it is currently not mentioned in the Composable Architecture Framework.

Another crucial aspect of the PBC platform is the design, deployment and running of the PBCs. Gartner sees that a change in a PBC Type would yield an update of all composed

applications that use such PBC Type. In NS, the updates of templates making up the elements are handled through expansion and rejuvenation. The new version of the element templates triggers a rejuvenation cycle, updating all instances where this element template is used. The final step is to deploy the application.

Gartner simplifies this concept with an example of Planning PBC used in two compositions [5]. The update of that Planning PBC would trigger the update of both compositions. The question is, what is being updated? Is it a PBC Type that underlies the Planning PBC? Is it the Planning PBC template or the code making up the PBC? Does this update result in one deployment of the Planning PBC shared by two compositions, or are there two deployments of the Planning PBC? In the former case, it would mean that the Planning PBC has zero customizations (even in terms of low and no code). Otherwise, it cannot be used in two different composite applications without putting extra differentiation logic into the Planning PBC or the composite applications. In the latter case, it would mean that running instances of Planning PBCs are not identical, and the difference must be managed on the Platform, resulting in different PBC versions. We already discussed the issues related to that topic.

The problem is that clear guidance is missing on how the PBC platform should handle this. The PBC Platform must also conform to the four NS principles, or it will not allow the evolvability Gartner portrayed in its Composable Architecture Framework.

VI. Validation By Focus Group

The previous section looked at the components of the Composable Architecture framework and tried to point out possible operationalization issues employing NS and business capabilities theories. To avoid our findings being considered too opinionated, we conducted a small experiment with students of the Antwerp Management School who had just been exposed to NS. The idea was to investigate how they look at a concept such as Composable Architecture once they know about NS.

Between September 2024 and November 2024, the Master in Enterprise IT Architecture (MEITA) students of the Antwerp Management School were exposed to Normalized Systems for 16 hours. The MEITA is an executive master, and students in the program already have many years of practical experience in IT Architecture. At the end of this period, they were given the Gartner paper about the Composable Architecture Framework and asked to read this document using their newly acquired knowledge of NS. All students got the reading material beforehand, were asked to read it, and were asked to discuss it in groups for thirty minutes and provide a summary of their findings in 5 minutes. In total, 18 students participated, split into four groups.

The first group had difficulty understanding the meaning of Packaged Business Capabilities (PBCs). They realised that the approach can only succeed if business and IT people define the PBCs. They expected more guidance on how to do this and foresaw evolvability issues, as no mechanisms were foreseen to adequately address SoC beyond business changes (what about technological changes?).

The second group tried to list the pros and cons of the framework. They found the usage of no/low-code with a marketplace of PBCs to be powerful. They considered the recommended usage of headless promoting technologies to be a good way to decouple UI and logic, adhering to SoC at least for those two concerns. Group one also struggled with understanding PBCs and claimed they had never observed it in their practices. They see the framework's application more in classifying your existing applications according to business capabilities and then use the Composable Architecture Integration platform to recombine existing applications.

The third group also struggled to understand PBCs. They compared the main characteristics of PBCs with the NS theorems. Modularity resonates with NS, but the framework has issues describing the level of granularity. They see SoC applied to some extent but insufficient to be NS compliant. They saw that the second characteristic, autonomy, would require SoS, but this is not mentioned. They make a similar remark about PBC orchestration. They conclude that the fourth characteristic, discoverability, misses the need for AvT and DvT.

The fourth group noticed that while NS is about change over time and minimizing ripple effects, PBC is about building fast. The characteristics of PBC are such that nobody in their right mind would not want them, but the framework insufficiently explained how to do it. On the other hand, NS tells you how to do it, and doing it the NS way requires significant effort. The group also identifies the PBC granularity uncertainty as a source of future issues. For instance, the sales business capability, packaged in a PBC, may or may not answer to the different sales business capabilities needs in large and complex organizations, opening the door for violation of the four NS principles quicker than expected.

The different groups struggle with similar issues. Once exposed to NS, one can ask more profound questions about how implementation should occur and whether a proposed solution has the characteristics it claims to have. It is important to note that this does not necessarily mean that NS is the optimal solution to operationalize the concept of PBC, as only a single methodology was evaluated in the focus group. Nevertheless, it does seem to validate that there is a need for a strategy to operationalize a PBC architecture through more concrete guidance, such as provided to a certain extent by NS.

VII. Conclusion

Application re-use is not just a long-forgotten dream of McIllroy, but a focus on many companies. The ability to reuse and recombine applications to support the changing business conditions of an expectation many CEOs have toward their CIOs. In recent years, the focus on functional reuse has been pushed aside by technology-focused integration patterns. Gartner puts functional re-use back on the map with its Composable Architecture Framework, where PBCs are the

essential building blocks of application landscapes. By using NS as an instrument of design and evaluation method for evolvable systems, we pointed out operationalization issues one might face when trying to implement the Composable Architecture Framework, or one might use it to critically evaluate providers of solutions based on it. A focus group was used to validate and balance our findings.

ACKNOWLEDGMENT

The authors thank Rudy Claes of Innocom for introducing them to Gartner's Composable Architecture Framework and conducting a brainstorming session about the framework and its evolvability. We also thank the Master in Enterprise IT Architecture (MEITA) students at Antwerp Management School (AMS) for their contribution as focus group.

REFERENCES

[1] G. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems: Concepts and Design Edition 3," ISBN:978-0-201-61918-8, 2001.

[2] J. Sun and Y. Natis. "Use Gartner's Reference Model to Deliver Intelligent Composable Business Applications," Gartner, ID G00720701, 2020 - refresh 2022.

[3] H. Mannaert, P. De Bruyn, and J. Verelst, "On the interconnection of cross-cutting concerns within hierarchical modular architectures," IEEE Transactions on Engineering Management, Vol. 69, pp. 3276-3291 2020.

[4] P. Huysmans, G. Oorts, P. De Bruyn, H. Mannaert, and J. Verelst, "Positioning the normalized systems theory in a design theory framework," Lecture notes in business information processing, ISSN 1865-1348-142, pp. 43-63, 2013.

[5] Y. Natis and G. Alvarez, "How to Implement Composable Technology with PBCs," Gartner, ID G00751018, 2021.

[6] H. Mannaert, J. Verelst, and P. De Bruyn, "Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design," ISBN 978-90-77160-09-1, Koppa, 2016.

[7] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," Science of Computer Programming, Volume 76, Issue 12, pp. 1210-1222, 2011.

[8] P.Huysmans, J. Verelst, H. Mannaert, and A. Oost, "Integrating information systems using normalized systems theory: four case studies," In IEEE 17th Conference on Business Informatics, Volume 1, pp. 173-180, 2015.

[9] A.W. Scheer, "The Composable Enterprise: Agile, Flexible, Innovative: A Gamechanger for Organisations, Digitisation and Business Software," ISBN:978-3-658-42482-4, Springer, 2024.

[10] I. Ivas, "Implementation of Composable Enterprise in an Evolutionary Way through Holistic Business-IT Delivery of Business Initiatives," In Proceedings of the 26th International Conference on Enterprise Information Systems, Volume 1, ISBN: 978-989-758-692-7, pp. 397-408, 2024.

[11] MACH Alliance, [Online], Available: https://machalliance.org, [retrieved: March, 2025].

[12] J. Barney, "Firm resources and sustained competitive advantage," In Journal of management, Volume 17, Issue 1, pp 99-120, 1991.

[13] T. Offerman, C.J. Stettina, and A. Plaat, "Business capabilities: A systematic literature review and a research agenda," In International Conference on Engineering, Technology and Innovation (ICE/ITMC), pp. 383-393, 2017.

[14] BIAN, [Online], Available: https://bian.org, [retrieved: March, 2025].

[15] NBility, [Online], Available: https://www.edsn.nl/nbility-model, [retrieved: March, 2025].

[16] SAFe Framework, [Online], Available: www.scaledagileframework.com, [retrieved: March, 2025].