Robust Digital Information for Physically Large and Complex SoS: A Maritime Design Case Study

Janica A. Bronson
IHB, NTNU
Ålesund, Norway
janica.echavez@ntnu.no

Henrique M. Gaspar
IHB, NTNU
Ålesund, Norway
henrique.gaspar@ntnu.no

Cecilia Haskins
MTP, NTNU
Trondheim, Norway
cecilia.haskins@ntnu.no

Ícaro A. Fonseca
IHB, NTNU
Ålesund, Norway
icaro.a.fonseca@ntnu.no

Jose Jorge Garcia Agis
Ulstein Group AS
Ulsteinvik, Norway
jose.jorge.agis@ulstein.com

***Abstract*—This paper aims to contribute to understanding how physically large and complex systems of systems (SoS) can be digitally modeled. Ensuring the modeled SoS accurately represents real-world systems is challenging. Due to the significant amount of interaction these models have with the actors using them throughout the design lifecycle, their reliability can often be unpredictable and typically degrade unless constant verification and quality assurance protocols are followed. This paper aims to shed light on ways to address this concern, particularly by gleaming into how information systems (IS) can be made robust to aid in making data models less vulnerable to change.**

***Index Terms*—robustness, information model, information system, data persistence, provenance**

I. Importance of Digital Information Robustness

Systems of Systems (SoS) are defined by INCOSE as a 'collection of components organized to accomplish a specific function or set of functions'. To be considered an SoS, individual components need to be able to maintain managerial and operational independence from each other [1]. In the case of physically large SoS, these components and subsystems are also typically highly connected and coupled. Therefore, modeling a physically large SoS is quite challenging [2] given the level of scale and complexity expected in retaining the independence but also the interconnectedness of the various subsystems.

For engineer-to-order (ETO) industries that can benefit from virtual modeling for greater design visibility, having the means to model physically large and complex SoS digitally is becoming a rising challenge. More specifically, the data model[1] of such systems needs to be critically considered. Unlike model-based engineering, an SoS model does not represent the system's behavior but rather the knowledge surrounding such a system. A data model is a shared information resource of a system that serves as a reliable basis or Single Source of Truth (SSoT) of knowledge throughout its life cycle. The concept is similar to the understanding of Building Information Models (BIM) in construction and Digital Mock-ups (DMUs) in aerospace, among others. In the maritime industry, specifically ship design and construction which will be the use case focused in this paper, this data model is closely tied to the asset information model (AIM) of the vessel developed in design and construction.

While the importance of a data model is related to the operational benefits that a consolidated SSoT solution will provide for information retrieval, there are other benefits of having an accurate data model. An accurate model can help transform a complex system's design methodology, especially in enabling better visibility of design risks in reference to specifications or design goals. Most importantly, accurate models reduce inaccuracies, information entropy, and noise related to knowledge about the system.

This paper focuses on ensuring that robust information systems (IS) are in place to maintain the accuracy of a system's data model. An IS collects, processes, stores, analyzes, and disseminates information within an organization for a specific purpose [3]. A robust IS is one that can accommodate and tackle change, whether imposed over time or as alterations to the system's understanding by various actors using the system. Maintaining a robust IS is crucial to ensure that the data model of complex systems remains effective in representing real-world equivalents.

To illustrate the critical elements that need to be considered in order to develop a robust IS for a large-scale and complex SoS, a case study is developed in this paper specifically for the change management of a subsystem in a physically large and complex SoS (i.e., a vessel). Additionally, the complications that arise in terms of delay and error propagation from the lack of robustness are also described.

II. Vulnerable Information Systems

Vulnerable information systems that are sensitive to interactions with the environment and human actors are prone to noisy data and risk inaccuracies in data modeling. In this

[1]In line with the data, information, knowledge and wisdom hierarchy (DIKW), knowledge discovery and data mining (KDD) and data-driven decision making (DDD) concepts.

section, we will explore the meaning of a robust and less vulnerable IS and the different types of change that may affect the data within an IS.

A. Robustness

The ability of a system to function correctly despite changes in its inputs or environment is called *robustness* [4]. For an enterprise IS, robustness is well-understood to be connected with the interfacing and interaction of the data model with the agents and actors using and transforming such data model in the IS.

An actor-focused IS is based on the hypothesis that most concurrent communication systems today that share information regarding an asset can be modeled, understood, and reasoned in correlation to the actors using such models. The basis of such a hypothesis is the premise that in order to ensure the scalability and robustness of information systems, the consideration of 'actors as universal primitives' for concurrent digital computation is necessary. Hence, tied with the understanding of robustness, assessing the IS' scalability and the level of concurrent data usage are critical [5]. With the actor-focused definition of robustness in mind, general properties of what is desirable for a robust IS need to be considered [5]. Notable properties are defined below:

1) **Persistence** - All information is gathered and organized in a systematic manner without losing the original data.
2) **Concurrency** - Work is processed interactively and developed simultaneously over time.
3) **Pluralism** - Information can vary greatly in type, with different sources often presenting overlapping and sometimes conflicting details.
4) **Provenance** - The origin or source of information is closely monitored and documented for accuracy and accountability, such that there is information provenance or a documented information trail.

The well-accepted understanding of an IS comprises six main elements covering *hardware* referring to the machinery and equipment; *software* referring to computer programs and the manuals that support them; *procedures* referring to the policies that govern the IS' operation; *people* referring to the actors using the data, the *networks* enabling the bridge between hardware and people, and finally the *data* or associated databases [6]. Additionally, an IS has historically been viewed as having different use cases that support the users of such systems in various ways. For instance, workers may focus on Transaction Processing IS, middle managers may need Management IS, senior managers may need Decision Support IS, and executives may require Operational and Strategic IS. Hence, overall, people are the most sensitive component of the system. Their usage, as well as their needs, inform and determine the system's success or failure. Therefore, it is essential to evaluate the vulnerabilities of the IS based on the interactions made by users with these systems.

Given the components and properties discussed in this section, a robust IS in this paper is considered to provide a means for processing information and carefully recording its provenance. That is, an IS supports the reduction in information loss, which may disrupt a development workflow.

B. Types of Change for Data Models

A data model can encounter different types of change. These can range from well-known or expected changes, as in the case of version control changes to the data model, or they can involve uncertain and unexpected disruptions. Responding to the latter is a critical feature of resiliency and is outside the scope of this paper. For this paper, the following types of change are considered:

1) **Version Change** - Change in the entire data modeling compelling a full version update. For example, when designing a system that involves variableA, the user changes the number assigned to such variable.
2) **Model Change** - Changes in the boundaries of the model, such as changes in the types of variables or attributes used to describe the model. For example, a user determines that variableA is no longer needed and removes this variable.
3) **Goal Change** - Changes in the model's reference, especially for verification and checking purposes. For example, a user determines that the model no longer meets certain thresholds, which changes the model's sensitivity to its goals. These changes can also include changes in the context of the system.
4) **Modality Change** - Changes in the means by which users can interface with the model, including the types of software and the modalities available to interact with the model. This may be related to software changes affecting how users format or interact with the model. For example, due to a switch from hard-copy design processes to online processes, users may be required to redefine the model to match the new digital format requirements.

III. Mechanisms to Track and Enable Information Change

In current practice, various industries have their own procedures to manage change within their own enterprise IS. To understand how these can be made more robust, we explore current practices for change management as well as various conceptual mechanisms and practical applications in this section.

A. Current Practice

In current practice, various IS and data models have variable degrees of receptiveness to change, depending on the industry. Ensuring that information or an SSoT model is built for changes is quite challenging and requires industry-wide acceptance. An example of an industry-wide protocol is GIT in software development, which was created to manage code revisions. GIT is a distributed version control system that tracks changes to computer files and enables collaborative and non-linear programming workflows. Outside software, this process is called Engineering Change Management (ECM),

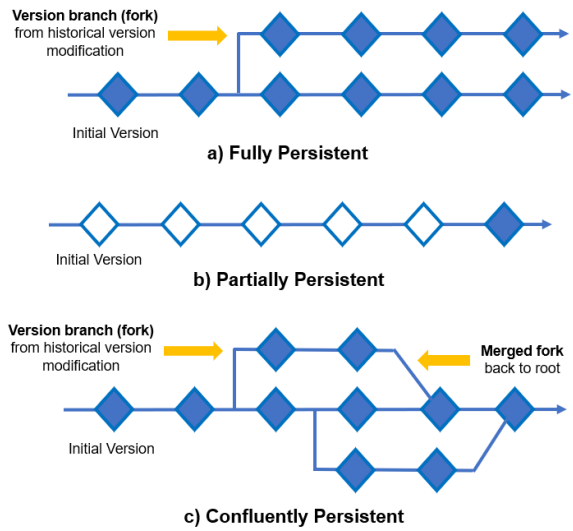

Fig. 1. Version Diagrams for Fully, Partial, and Confluent Persistence

which is an operational-focused and often manual approach to managing changes. Currently, these ECM methods are highly cumbersome and time-consuming, but they are the only alternative for large ETO companies. Without these practices and/or funnels, there is no way to harmonize design changes in some companies; instead, the proliferation of various data models that are distributed among actors involved in the design development process is likely.

To determine how to address this problem from the data modeling perspective, we gleam into the topic of data structures. In computing, a *persistent data structure* allows holding multiple versions of a data structure at any given time, enabling users to view, access, or even modify previous versions and faciliating historical auditing. Persistence is categorized by permitted operations on past versions [7], as described in the following section. Figure 1 illustrates these differences, where the diamonds represent versions of a document. Colored diamonds represent read-only or immutable versions, and unfilled diamonds represent read-write versions.

B. Types of Persistence

1) ***Full Persistence****:* Fully persistent data structures enable access or queries and modification of all historical versions without modification restrictions. This is displayed in Figure 1a, where a new version branch is created when historical versions are revisited and modified.

2) ***Partial Persistence****:* Partially persistent data structures allow access or query of all historical versions but not modification. Only the newest version can be modified, making historical versions immutable, as displayed in Figure 1b.

3) ***Confluent Persistence****:* Confluent data structures enable modifying historical versions while merging them with existing ones to create a new version, as shown in Figure 1c. An example of a fully confluent persistent structure is that of GIT, which supports the 'merging' of version branches (forked or cloned) back into the source code. In this case, users need to 'commit' before merging and providing a label for the changes committed to the source code. Due to the number of versions being developed in parallel, tools such as GitKraken that allow a visual graphics user interface (GUI) to understand which parties are committing to the root are helpful.

C. Conceptual Applications

In software engineering, specific advanced algorithms and methods are applied to achieve such persistence [8]. This section discusses these further to understand how persistence can be achieved conceptually.

1) ***Fat Node****:* A data structure can be made more persistent by incorporating a modification history of every node of information. In this context, a node is a basic data structure unit that may be linked to other nodes. Thus, this method is often given the moniker 'Fat Node' as every new node contains imprints of its value at any previous time. For a fully persistent structure, each node would include a version tree and not simply a version instance.

Unfortunately, using this method to access imprints in time is highly cumbersome. Accessing the desired version of a node would take an enormous amount of time to search, depending on the number of modifications made. In terms of partially persistent models, this is applied by using a binary search that can help to access a version given a timestamp that the user must be aware of.

2) ***Path Copying****:* This alternative method addresses the problem by 'copying' the node before changing it. After that, any changes to the old node must be cascaded to the new node. The root is, therefore, the node that no other version points to, but the cascading changes will always reach the root to some extent.

Accessing time is not as intensive in this structure; however, space is easily consumed, given that a copy of the entire structure is duplicated for every new change.

D. Implementation

The previous two methods provide persistence at a data level, which can typically be implemented in terms of user-developed data structures, whether in lists or arrays. In coding these structures, persistent implementation is often done in languages such as Clojure, Haskell, and JavaScript, and more explicit applications are done for Python or other languages with specific immutable data structures.

However, in terms of unstructured data in databases, as in enterprise data context, persistence may be enabled manually following the same logic described in the previous section. For example, a file can be made read-write or read-only to ensure it is partially persistent. The simple undo/redo functionalities are also an example of persistence in text editing.

In terms of implementation, the limiting constraint is related to storage. Storage mechanisms are, therefore, critical to facilitating provenance and persistence. In most applications, various file states can be stored within a computer's memory, typically in Random Access Memory (RAM). However, this

type of memory is often volatile, despite its speed advantages, and can be affected in instances of power outages or similar. On the other hand, disk-based storage of the use of persistent storage systems such as hard drives provides a less volatile means to store data at a state that cannot be overridden unless done manually. These two are further described below.

Hence, in terms of implementation, persistence can be applied through manual and automated means but constrained by storage. For this reason, computer systems design still relies on defining memory hierarchies such that storage device preferences are arranged based on access speed and size. This enables more frequently accessed data to be available readily compared to data that is used less often. Additionally, these considerations affect overall data dependability and security (i.e., data integrity) such that data is still available despite unforeseen and external circumstances.

However, it is important to note that capturing as much data to have a complete view and provenance of the information generated is not solely dependent on storage but also on the understanding of the data ecosystem, including the platform and tools used for data management. These components include hardware and operating systems. Additionally, persistence is just one aspect of database design, and designers should also consider the performance of the database.

These other considerations make scalability a huge challenge, especially when it comes to implementing persistence and provenance into tangible storage and memory. For instance, a single-ship program may require up to 10 gigabytes of data as of the early 2000s [9]. According to Wang (2015), to get any fine-grain data provenance will require space that is several times larger than the actual data being processed. In the case of enabling this, there is a large number of collection overhead that could be expected. Hence, full persistence and complete provenance often come at a steep cost. Handling provenance and persistence at scale is a rising field that now merits the use of the term *veracity* specifically for the provenance of Big Data [10]. New storage mechanisms are also being considered, such as cloud storage, to enable provenance in today's cloud-centric environment [11]. In the context of a distributed cloud environment where there is high-velocity and high-volume data, centralizing the metadata that captures this provenance and persistence is still a relevant obstacle today [10].

IV. Case Study

In enterprise applications, varying file formats and multiple actors interacting with the IS make the efficient application of persistence not often easy to implement. For most industries, such as manufacturing, having a streamlined process and development pipeline can circumvent change management challenges. Unfortunately, designs are often bespoke for ETO industries and may vary from project to project - making change management an integral but largely reactive process. To describe this case, the maritime industry, specifically ship design and shipbuilding as a complex ETO industry, is used.

A. Ship Design ECM

Early-stage ship concept design is fast-paced and iterative, requiring quick change management and constant validation. Multiple teams and departments often concurrently manage several steps, and various versions of a concept (whether the entire ship or subsystems in the ship) are matured in parallel to best explore the design space.

For this study, we focus on early systems design in the concept stage, primarily on the selection and specification of a propulsion system within the ship. There are various components in a ship, covering ship propulsion, electrical systems, and mission equipment, among others. To some extent, these systems can be highly independent of one another, especially in the case when modular shipbuilding practices are applied, and redundancies are built such that these major subsystems can operate independently. These steps, for the design and management of the design of a single subsystem (i.e., propulsion system) in the SoS, are oversimplified and inspired by the work of Gale (2003) [12]:

1) Validate the top-level ship performance requirements and develop second-tier requirements
2) Establish preliminary system dimensions and configuration constraints
3) Select major ship propulsion components and equipment
4) Quantify ship performance
5) Evaluate costs
6) For second-tier performance requirements, derive performance thresholds from the higher-level ship performance requirements
7) Develop and evaluate alternative system configurations
8) Select system configuration from alternatives
9) Complete engineering work on the selected system, and finally
10) Develop propulsion system specifications and drawings

Figure 2 shows the theoretical versioning sequence of these steps applied and the potential vessel design iterations developed in the process. In this figure, the change process is illustrated where the numbered diamonds describe the edits made to accommodate the steps as numerically ordered above. In contrast, the filled diamonds are the versions developed after the modifications are complete. In this illustration, the data model is simplified to be an SSoT database with unstructured data covering the shareable and consolidated data from supporting information and modeling work. The horizontal axis represents the design maturity, although not representative of any company's engineering timeline. The different lanes visualize the actors or departments working on these versions, including the engineering team focused on supporting information, the integration team working on the data model, and the outfitting and modeling team working on the 3D model of the vessel. The various interactions between these departments are illustrated via the exchange of data from one lane to another. For example, version A of the SSoT database or data model is developed after data from various preliminary analyses and 3D modeling are integrated together. A new version, version

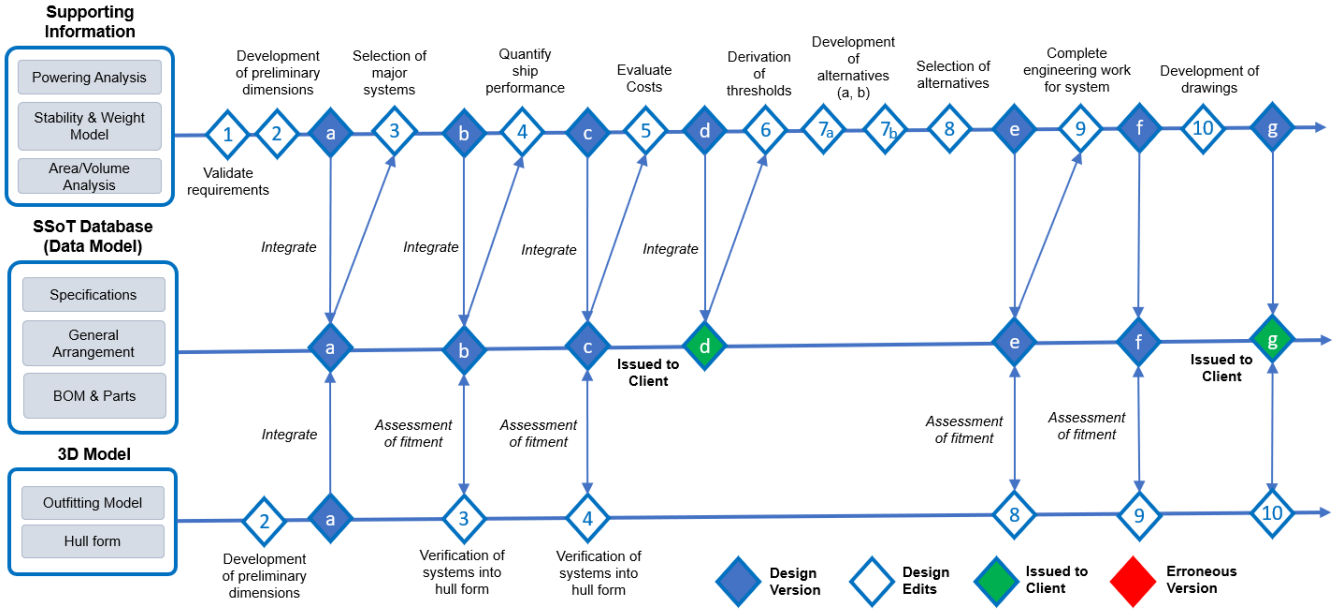

Fig. 2. Ideal Theoretical Version Changes for an ECM

B, is then developed after major components are selected by the engineering team and then verified by the outfitting and modeling team. These departments, therefore, have to manage multiple revisions and ensure that data from different analyses, models, and calculations are incorporated into only the latest version of the data model.

These departments are simplified for this visualization; however, a company can have various overlaps between such teams depending on their organizational structure. The demonstration also assumes that changes are performed linearly, not concurrently; that is, a version in a different format is strictly not modified by other parties until a version update is complete. Hence, challenges that arise from parties and personnel interacting with the information over time are not completely covered in this case.

Unfortunately, this change process is ideal and does not display the possible complications arising from errors, delays, and actors not following proper protocols. Figures 3 and 4 illustrate these dilemmas.

Figure 3 shows an example scenario of intermediary versions developed when errors are made into an internally issued design version. As design revisions based on error correction are common in engineering processes, this type of ECM disruption is to be expected. It is, therefore, possible to have multiple loops of intermediary versions, which may be challenging to track and require more time than expected, placing pressure on executing the subsequent task.

On the other hand, Figure 4 shows the effects of delay on the process. For example, in the event that task 3 in 3D modeling is delayed in verifying the systems fitment in the model, and other departments work with the outdated version (such as version a), then an erroneous version of the design with mismatched degrees of maturity can be developed.

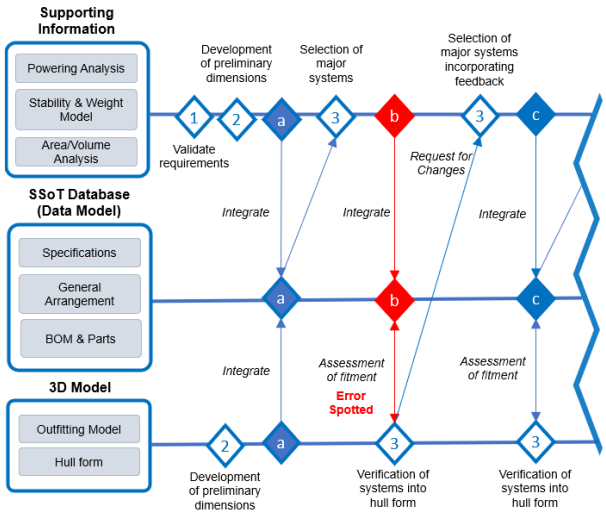

Fig. 3. Intermediary versions due to errors

Unfortunately, this erroneous version will likely be carried forward as the design matures. These two cases are examples of the potential propagation of error and the sensitivity to delays of current data models due to the manual and highly process-oriented ECM methods today.

B. Persistence Applied to this Case

As we are only drawing inspiration from computational concepts, it is essential to keep in mind that a persistent data model is realizable when we have a data structure encoded in machine-readable language or format. For example, to arrive at confluent persistence, modifications should be mergeable, which is achievable when changes are made in a more or

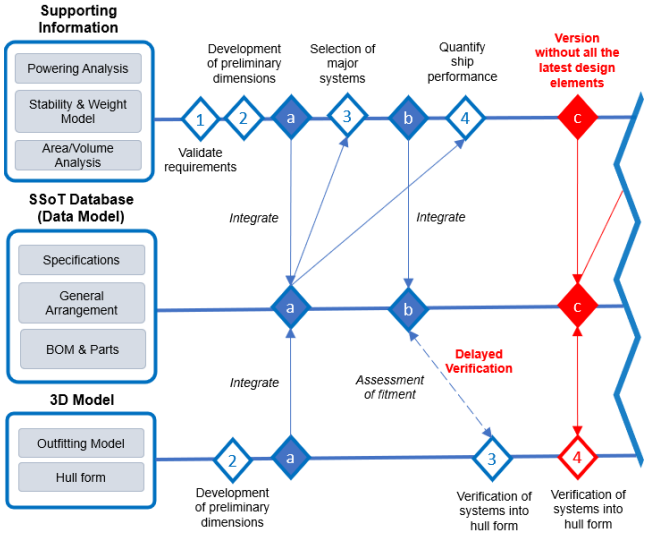

Fig. 4. Errors due to delays

less similar format. Unfortunately, most data repositories are highly unstructured, such as file systems that will include a combination of text and visual elements. Hence, for the application of persistence in this case, it is assumed that such a structure with consolidated ship data exists and serves as the 'root' file from which version trees can be forked.

Provided that we are using a persistence data structure described, the methods of path copying and fat node may be conceptually applicable. In order to avoid relying on periodic data saving or snapshotting, we also rely on disk-based storage when implementing the solution.

As previously described in terms of the elements of an IS, having hardware, procedures, and motivated persons to implement such robust IS and ECM processes is paramount. For instance, similar to GIT practices, requiring a commit before merging with the root is imperative, as discussed; otherwise, a merge should be rejected at risk of creating erroneous edits. Figure 5 illustrates what this change management process would look like if all these components were to come together:

- **IS that has persistent data model**: In this case, a fat node application is considered, where old versions are stored whenever a new version is created. This type of persistence is partial but provides a way to view older versions and work only on the latest version. Hence, this creates a 'root' file or source, which is always going to have the latest version, removing any disruptions that may be caused by persons manually picking the wrong version to work on. Any delays will be transparent as changes will be instantly reflected in the same root. Additionally, this guarantees that designers and even clients are always viewing the latest design version.
- **Hardware to enable better storage for such persistence**: Having a fat node would require a significant amount of storage to ensure past versions are never lost. Hence, hardware with enough disk space and reliable database infrastructure is considered.
- **IS with procedures in place to enforce such collaborative change management**: In this case, confluence can be achieved manually if users follow strict steps and rules in implementing changes back to the root. Such changes should be applied with verifiable reasons and after thorough quality assurance of engineering or modeling work.

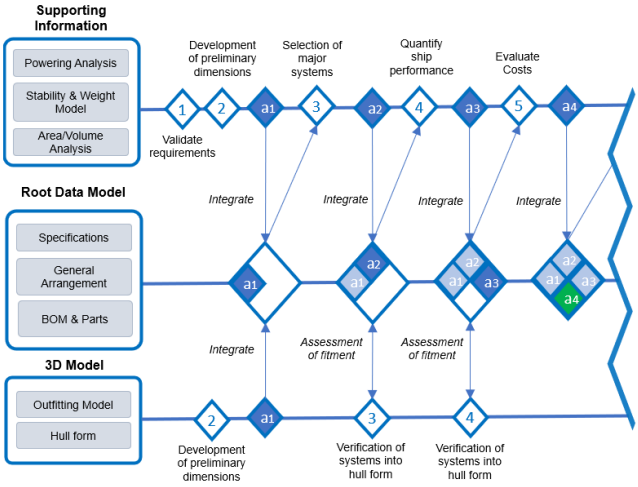

Fig. 5. Partial Persistence applied to this case

V. Robust Information Systems for Complex SoS

The example provided shows the dilemma of version and change management for a single system in a vessel. The partially persistent fat node for version A in Figure 5 is only one part of a more extensive physical and functional vessel system, as illustrated in Figure 6. At the same time, within the enterprise context, the more extensive vessel system it is part of is among many other projects that have their own individual ship data models. Where physically large and complex SoS is involved, the challenges related to version delays and errors from subsystem design can propagate rapidly and compromise the understanding of the SoS as a whole.

As discussed regarding actor-focused robustness, the robust IS characteristics described by Hewitt (2010) will need to be carefully assessed for concerns in scalability, especially in handling technical limits such as storage. Robustness can be more challenging with more dynamic SoS, where there is greater concurrency and data pluralism [5].

VI. Conclusion and Next Steps

This paper reviewed the concept of robustness in considering the IS of large-scale and complex SoS, especially in ensuring that SoS' data model remains accurate amid change.

While the conceptual understanding of a robust IS at a focused subsystem level is explored in the case study, the implementation and development of such IS is still underway.

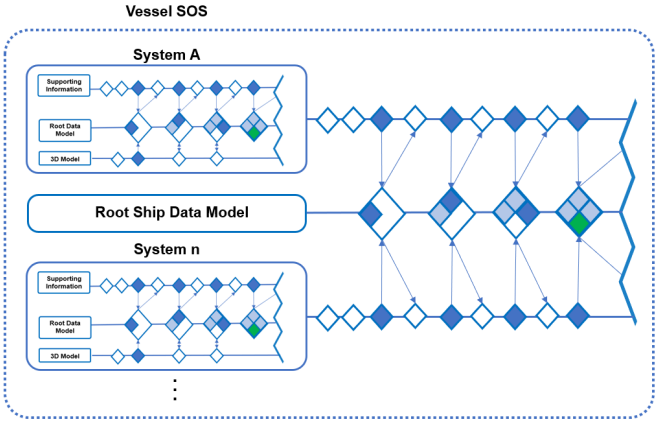

Fig. 6. Information Exchange within SoS

For the case described herein, the Smart European Shipbuilding (SEUS) Project [13] is currently tackling the development of such a smart platform that takes change management into consideration. The project's goal is for these tools to ultimately be used within yards and design firms, facilitating more reliable and collaborative design practices.

The authors hope that awareness about more robust IS to enable digital change management can help sectors that work with physically large and complex SoS arrive at a more efficient means to retain the accuracy of information models related to these systems.

REFERENCES

[1] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/(SICI)1520-6858(1998)1:4¡267::AID-SYS3¿3.0.CO;2-D

[2] S. Page, *Understanding Complexity*. Pennsylvania: The Teaching Company, 2009.

[3] R. Rainer and B. Prince, *Introduction to Information Systems: Supporting and Transforming Business*. Wiley, 2022.

[4] M. Burgin, "Robustness of Information Systems and Technologies." WSEAS Press, 2009.

[5] C. Hewitt, "Actor Model of Computation: Scalable Robust Information Systems," 8 2010.

[6] K. Laudon and J. Laudon, *Management Information System*. Pearson India, 2016.

[7] E. Demaine, "Definitions of Persistence," 2 2012.

[8] D. Glasser, "Persistent Data Structures - Introduction and motivation," 2006.

[9] R. I. Whitfield, A. H. Duffy, P. York, D. Vassalos, and P. Kaklis, "Managing the exchange of engineering product data to support through life ship design," *CAD Computer Aided Design*, vol. 43, no. 5, pp. 516–532, 5 2011.

[10] J. Wang, D. Crawl, S. Purawat, M. Nguyen, and I. Altintas, "Big data provenance: Challenges, state of the art and opportunities," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 10 2015, pp. 2509–2516.

[11] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Provenance for the Cloud," in *Proceedings of the 8th USENIX conference on File and storage technologies*. San Jose, California: USENIX Association, 2010, pp. 14–15.

[12] P. Gale, "THE SHIP DESIGN PROCESS," in *Ship Design and Construction*, T. Lamb, Ed., Jersey City, 2003, vol. I.

[13] H. Gaspar, H. Koelman, and J. Jorge Garcia Agis, "Can European Shipyards be Smarter? A Proposal from the SEUS Project," Tech. Rep., 2023.