A software engineering perspective on digital twin: many candidates, none elected

Antoine Beugnard

A software engineering perspective on digital twin: many candidates, none elected.

(Position Paper)

Antoine Beugnard
IMT Atlantique - Lab-STICC
CS 83818
F-29238 Brest, France
Email: antoine.beugnard@imt-atlantique.fr

***Abstract*—The digital twin (DT) is a central idea in the digitization of society. The goal of this position paper is to acknowledge this idea as a paradigm and discuss how current implementations are far from software engineering standards. I propose a consideration of 8 aspects of the DT: the paradigm, its concretization as a component, its variability, its building process, its description with models, type and instance differences, its relation with simulation, and its composability. I conclude that although many digital twins claim to be such, none of them fulfill the definition as expected. This position paper provides food for thought.**

I. Introduction

Digital twin is a relatively new concept introduced by Grieves in 2014 [1] (previously named "Mirrored Space Model" [2]) and applied in many fields. The software community has naturally seized on this concept since it involves digitization and relies on many software and models.

Many review papers (e.g., [3]–[8] which reference 74, 303, 146, (356 + 49), 136, 75 articles, respectively) have been published, and dedicated conferences have been organized. Digital twins of many kinds have been presented, developed, and used: digital twins of products, of organization, of humans, of factories, of cities, and even of an entire country (Gemini project [9]). Many definitions have been proposed [10], as have many architectures for DTs.

The objective of this article is to offer a software engineering perspective on digital twins, drawing a parallel between digital twin and software.

It is worth noting that while the digital twin is about digitalization, it is more than just its software. Beyond the fact that a digital twin is connected to a reality (usually a cyber-physical system, or more generally a socio-technical cyber-physical system that includes humans), a digital twin can have its own hardware, sensors and physical resources. Thus, a digital twin is a cyber-physical system (usually) connected to another cyber-physical system, where the first is the reference system (RS) and the second is the digital twin (DT). The purpose of the second is to reflect the first and help users to interact with the first through enriched information (e.g., augmented reality), dedicated interfaces, and specific tools.

The remainder of this article draws a parallel between digital twins and several software engineering aspects: object-oriented programming (II), software components (III), variability (IV), process (V), models and metamodeling (VI), type and instances (VII), simulation (VIII), and composition (IX) to support analysis and presentation of several important software challenges.

II. Digital twin and objects

Historically, humans started building complex hardware systems far before building software. Then, programming appeared and software became a field of study in itself. To control hardware, of course, but highly focused on computation, data representation, algorithms, and compilation. Connection to the real world remained confined to *drivers* or approaches where *real-time* is central. Then, cyber-physical systems were identified as complex systems where both hardware and software are interleaved. These CPS have many variants: System of System (SoS), Socio-Technical Systems (STS) Systems when humans are also considered as part of the system, etc.

Research and practice on computation lead to the identification of many paradigms: functional, procedural, rule-based, object-oriented, and so on. All these paradigms have their strengths and limitations, their field of choice. The immateriality of software allows a great freedom on ways of thinking. Hardware is more tied to reality with physical constraints. Physical systems are naturally decomposed in subsystems, parts, devices, etc. Hence, a "natural" decomposition emerges.

It appears to me that the digital twin is in fact a paradigm; a way of thinking and building cyber-physical systems. The objective of this system is not only to control something, but also to present it to users, to monitor it, to simulate it, to improve it, to understand it, and so on. Quite naturally, digital twins are a way of achieving these objectives with an expected property of composition. Digital twins should be able to collaborate (to be composed) in order to build more complex digital twins of more complex systems.

The goal of object-oriented programming was very close. First evangelists of OOP insisted on the "natural" mapping between real objects and programming objects, making programming "simple". In fact, as everybody know, it is more subtle. But the abstraction provided by objects is fruitful [11]. Programming objects are often mapped to abstractions not

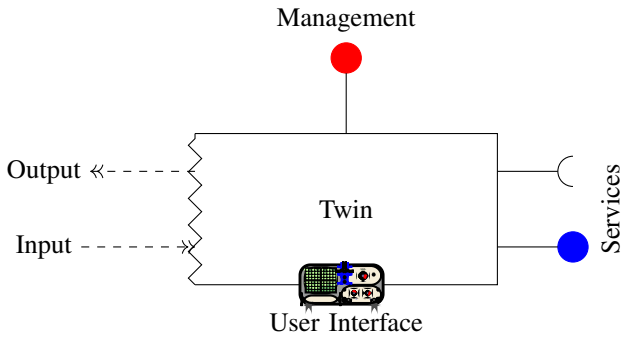

Fig. 1. Digital twin as component with its interfaces.

actual or physical things. A question emerges: May we have to develop abstract digital twins?

I think yes. Imagine the digital twin of a communication between two humans. We may have to develop two human digital twins (HDT), one for each human, probably a digital twin of the environment to collect information about the context (noise, distance, etc.) and why not an interaction digital twin that represent the properties of the interaction we are interested in. The idea is to reify an abstraction that is of interest, and where computation has to be done on selected data.

Another paradigmatic question is whether we may or may not have several DT for the same reference system? Again, I think yes. Digital twins have purposes and may manage different data (levels of data). Then, a Reference System (RS) could be connected to several digital twins. For instance, the digital twin of the RS manufacturer, and another one for its owner. We may imagine that insurance or user digital twins may also be developed. Ideally, a single digital twin, *the* digital twin, should be developed for all purposes, however, this can only be a theoretical point of view and not a pragmatic one. Then, the question of synchronizing multiple digital twins or not, rises.[1]

To conclude this section, I believe we must consider digital twins as a paradigm. This helps us build, or more precisely *organize*, complex STS (socio-technical systems). In that sense, literature is right to use the term digital twin and current digital twin development can justifiably be called digital twins.

To go further, as a software engineer, digital twins could be made concrete as something close to a software artifact: a component.

III. Digital twin and components

Object orientation, as a paradigm, has been very fruitful. Many programming languages, methods, tools, and theories have been developed. Among good properties of objects, we may list: encapsulation (responsibility of internal state management, interface of services), classification (levels of abstractions, inheritance) and reuse. Modularity, low coupling, strong consistency were previously identified and other paradigms already gave adequate solutions (for instance functional programming). However, the OO paradigm has limitations: they have bad composition properties and they have a very low-level management process (create/use/garbage). Software architecture and software components have tried to improve this situation. Software components bring composability as a primary objective. Services such as connect, disconnect, start, stop, resume become parts of the management. In order to realize services, the interface has been enriched: required services became explicit and a management interface too. This gives birth to many component models such a OSGi, .NET, CCM, Fractal, ...

Concerning digital twins, the current state of the art gives only very partial responses. There is no real boundary, there is no real interface (neither offered nor required, nor with a high level of contract) and there is no management interface. This lack of concretization makes digital twins poorly composable, poorly reusable.

I propose to build digital twins as software components (figure 1). An explicit boundary must be identified. Attached to the boundary, an interface to the Reference System shows inputs and outputs[2]. Then, depending on the digital twin purpose, several interfaces[3] to other components can be provided (blue circle) or required (open cup). A user interface (bottom of the figure), with a dashboard and a control panel for example, may also be furnished. Finally, a management interface must be provided for the configuration, deployment, running, monitoring, etc. The boundary is important since it encapsulates/defines all what constitutes the digital twin. Interfaces are important since they define the way the digital twin can be used (inputs and outputs, offered and required services) and managed.

To go further, and since digital twins are made of many different sources of data and programs, I propose a loose definition of their internal architecture (figure 2). A digital twin is composed of 4 sets:

1) A set of drivers whose responsibility is to manage the connection to reality (Core)
2) A set of data repositories whose responsibility is to store (or give access to) all data produced during all lives (see definition below) of the digital twin.
3) A set of services used to monitor, analyze, explain, simulate, ... providing and requiring interfaces.
4) A set of management services (that have to be defined)

The *core* is the closest layer to the RS. It contains sensors, actuators, drivers, operational (raw) data storage. It can be called the "edge" layer. For performance sake, monitoring processes, anomaly detection, and possibly other services can be installed here.

[1] I imagine that, we, as humans, are developing many *neural twins* of parts of our environment... and we try to share or build consensus of description by talking, writing and doing together.

[2] An abstract digital twin may have no such interface if it interacts directly with other twins and gather data, information or knowledge, indirectly, from other digital twins.

[3] I do not detail the nature of interfaces, they can be synchronous, asynchronous, synchronized, specified by contracts, etc.

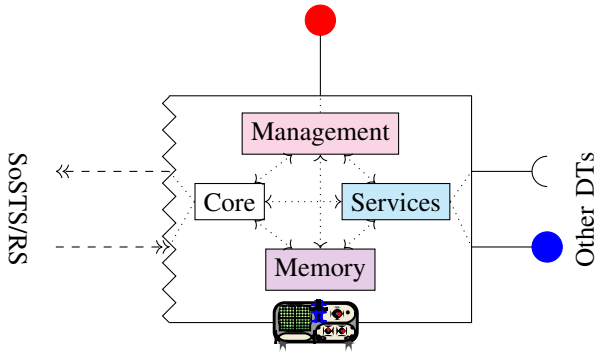

Fig. 2. Insight of digital twin internal.

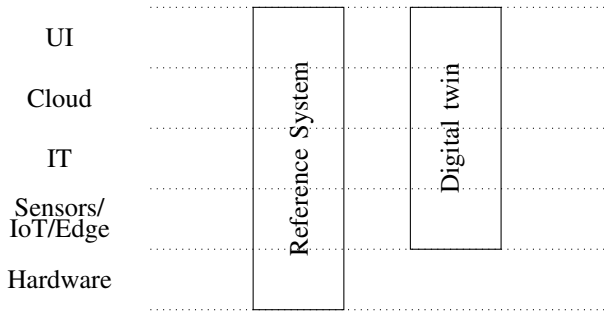

Fig. 3. A digital twin crossing layers.

The *memory* is the long-term memory layer. It can contain many kinds of information: possibly persistence for the core layer, but also information build from operational data and more sophisticated information called knowledge. The hierarchy of data (data, information, knowledge) can be stored in a multi-layer data storage : data in the edge, information in the fog (IT), and knowledge in the cloud.

The *services* gather all computations available: human interfaces, analysis, simulations, or predictive tools, etc. They can be implemented at any level (edge, IT, cloud).

The *management* package gathers all scripts, configuration files, installation and deployment recipes to manage the DT. It provides the glue to connect, deploy, and clone the DT and its services. They may be implemented at any level (edge, IT, cloud).

Figure 3 shows how a reference system and a digital twin cross layers. I make the assumption that a digital twin as no specific hardware, but can have sensors (hardware with software). Both may have, or not, parts at the IT or cloud level, and both can have, or not, user interfaces. UI are separated from IT and cloud because of their specific interaction with users.

To give an idea of what the management interface should provide, we have to imagine the life cycle of a digital twin. Figure 4 shows a simplified life cycle where the main steps of a digital twin life appear:

- status change operations:
  - install (prepare the configuration from development state to *installation* state)
  - connect/disconnect (connect to an existing CPS - or simulation environment to reach *instance* state)
  - start, pause, resume, stop (in *life* state)
- observation and query operations:
  - query (introspection)
  - view (run a specific monitor)
  - launch tool (data analysis, data enrichment, etc.)
  - archive (disconnect and prevent any future configuration)
- simulation operations:
  - clone (install another instance) (followed by a connect)
  - simulate (run an instance within a simulated environment)
  - replay (replay a stored history)

Note that some operations require information with a high level of abstraction such as an environment. High-level models probably have to be designed. We come back to this aspect in section (VI).

The service that makes a digital twin, a complete digital twin, is to my opinion, the **clone** operation. This operation is used when simulations are needed (to accelerate time, to optimize process or physical arrangement, etc.). The obtained clone may have to be modified (for studying variants) and connected to another real environment, or to a virtual environment.

Ideally, a digital twin component encapsulates its data, information and knowledge. That is, a digital twin is responsible for the access and delivery of information and thus controls accesses. However, implementations may release this strong assumption; it is then the developer's responsibility to check properties such as privacy, data protection, and so on.

To conclude this section, we can consider a digital twin as a component. This helps us build, or more precisely *assemble*, complex STS (socio-technical systems). In that sense, literature rarely refers to digital twins (seen as software components) and current digital twin development can hardly be called digital twins as they do not compose. For instance in [8], Boyes and Watson wrote "Although a significant volume of literature has sought to define the concept, there appears little agreement on the composition of a 'digital twin'." In another article [12], Michael and Wortmann build digital twins by software component assembly but nothing specific to digital twin management operation is proposed. Authors reuse a classical MAPE-K loop for self-adaptive systems, and externalize digital twin data in a data lake.

Now that a digital twin is considered as a component, the next question is how to deal with its variability?

IV. Digital twin and variability

Until now, we have considered a digital twin as a cyber-physical system connected to another CPS, that can run simultaneously. This is the instantiated and running version of

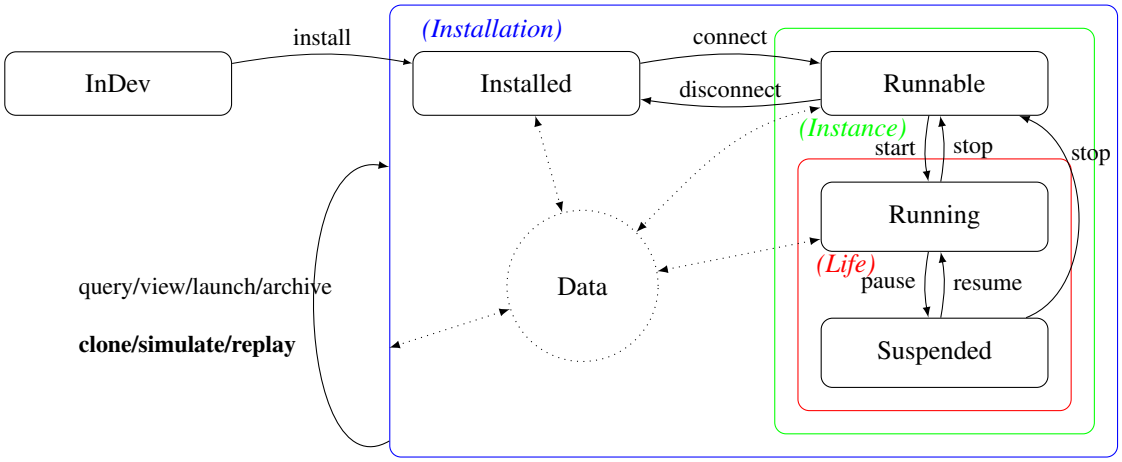

Fig. 4. A possible digital twin life-cycle.

a digital twin. However, before this running version, a digital twin exists in different forms. For instance:

- As developed (source code on the shelf, hardware specification)
- As installed (deployed source code on the destination infrastructure - partial configuration, selected hardware)
- As configured (fully configured code and hardware)
- As running (the configured version in operation)
- As expected (simulations - future or alternate configurations)
- Archived

At each stage, many variations are possible with different hardware, tools or configurations. During its lifespan, a digital twin may evolve by adding new models, new points of view, new dimensions, new tools. We come back to this process aspect in the next section (V).

The many faces of a digital twin, and the possibility of being instantiated several times, raise the question of identification. This is an important point since data are collected in a specific configuration, a specific context that had to be kept for the sake of analysis consistency. If the reference system is important, it is not sufficient. As a solution, I propose a naming scheme like: NameOfReferenceSystem:InstallationVariant:Instance:Life (see states in figure 4)

All this leads to the need of a powerful version management system to manage the digital twin and its variants. Git connected to variability models could be an idea as suggested in [13]. The git tree could encode the variants in their different levels of abstraction: from the fully configured running instances to the totally abstract description where only configuration patterns are identified without any values.

Now that a digital twin is considered as a component with many variants, the next question is how is it built?

V. Digital twin and process

The target artifact being identified, the question is now: what is the build process? The answer depends on a few factors: the need for diversity in implementations and usages.

In fact, one specificity of digital twins is the heterogeneity of its parts. A digital twin is a cyber-physical system with all its complexity. Drivers and tools required to the connection to the Reference System can be very different (sensors, IoT, controllers, etc.), techniques to store data can also be diverse (relational database, time series database, streaming storage, simple files, etc.) and many tools for monitoring, presenting, simulating, analyzing data, predicting, etc. can be reused.

Another specificity is the diversity of usages as shown by the digital twin consortium in its *Capability Periodic Table* [14]. This diversity of usage has obviously an impact on the diversity of implementations.

The last observation is the digital twin lifespan that implies that usages, tools and needs evolve during a long period of time.

Facing this diversity and durability, the typical software engineering answer for development is iteration. Each iteration improves the digital twin with refined aspects, additional analysis tools, new facets, etc. Following this, R. Parmar et al. identify five principles in [15]:

Principle 1: Start with what you have
Principle 2: Set the data free
Principle 3: Move the digitization frontier
Principle 4: Seek new digital opportunities
Principle 5: Increment the models

The first principle is summarized in the sentence, ”a full audit of the existing sensors, connections, data, and analytical models [of the organization][4], suppliers, and complementors, and then using those resources to create the initial entity model of the digital twin.” I would add tools, behaviors, and algorithms to cover the dynamic aspects of the reference system. The idea here is to collect and select what is inside the digital twin boundary and outside. What is managed and what is not. Building the digital twin becomes the realization

[4]brackets are mine.

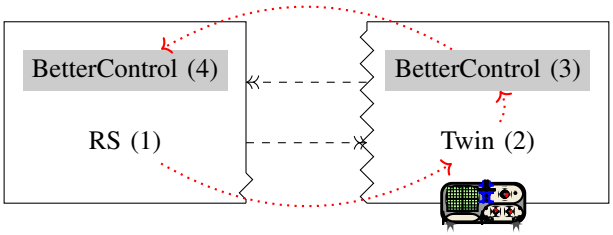

Fig. 5. Reference system and its digital twin: improvement loop.

of services on top of all the information collected: services for installing, configuring, running, simulating, and so on. As a software engineer, this resembles the way large applications or infrastructure are managed. This resembles *DevOps* tools: generic (shell script, makefile), or dedicated (Ansible, Chef or Puppet).

The second principle relies on the assumption that "the more the data are set free, the greater the potential scope of a digital twin." I would add, beyond data accessibility and all the ethical implications which are beyond the scope of this article, the importance of data classification, data organization, and metadata. All data produced must be labeled with all the details of production (when, how, who, for what, etc.) in order to facilitate access and analysis. If I agree on principle, we must recall that data are related to privacy (for human), to intellectual property (for companies), to secrecy (for countries) and must have specific access policy. I remind that encapsulating data makes the digital twin responsible for accesses.

The third and fourth principles invite to "focus on digitizing all kinds of assets, processes, and interactions and ensuring that the resulting data flows are compatible with the information, context, and impact models that comprise the digital twin" [not just digitizing physical assets] and "to identify and execute new digital opportunities to further extend the reach and coverage of the digital twin." These principles are consistent with our first observation considering digital twins as a paradigm for building new complex systems.

Finally, the fifth principle "requires an active extension of the digital twin rather than simply maintaining and updating the information, context, and impact models." This, again, meets software engineering principles were software and applications must evolve to remain of interest.

All these considerations lead to a dynamic evolutionary process close to existing iterative development processes that allow continuous improvement.

For instance, in figure 5 we show a development cycle with four steps. First (1), the reference system is identified and, second (2), a twin is developed. Then, a better control (3) improves the twin thanks to powerful simulation or analysis tools. Stopping the process here would be accepting the risk of coming back to the previous, not as good control, in case of digital twin halt or disconnection. I therefore propose to close the improvement loop by integrating the better control (4) into the reference system[5].

In [8], Boyes and Watson claimed that "[t]he concept of direct control of a physical entity by its digital twin is a fallacy. If a digital twin is subsumed into and becomes an essential element of the control system architecture of a physical entity, then it has ceased to be a digital twin and is now a subsystem forming part of the physical entity." Well, I guess it is a matter of interpretation. Remember that whatever the implementation of a digital twin is, it remains a model. That is, an approximation, a set of views on a reference system. But it is also more since results of analysis, predictions, simulations of variants are parts of a digital twin. It is also possible to enrich (extend) the control of an RS through a digital twin as shown in figure 5. The reference system and its digital twin become a couple of entities that evolve together.

Note that the reference system must be autonomous (i.e., being able to work properly independently). If a so-called digital twin cannot be removed (unplugged) from its reference system, it means we do not have a digital twin, but a part of the reference system itself. Conversely, the digital twin cannot operate without a reference system that feeds it with data[6]. It cannot operate, but it can be used, as an archive for example, or to compute statistics or predictions from previously stored data.

As software engineers, we are interested in languages for describing all digital twin aspects. That is interfaces (services and contracts), configuration (to the reference system or its data feeder), user interfaces, contents (for deployment and operation implementations), etc.

VI. Digital twin and models

Cyber-physical systems have many facets. Many dimensions could be described: shapes, locations, physical properties, data properties, time properties, behaviors, costs, energy, documentation, requirements, ... All these dimensions have been studied and tooled with dedicated software and models. Often, independently. One major challenge is to make all these tools and models cooperate smoothly... in duration. What are the data or information that can be changed?e.g. for another instance of the digital twin, for an optimized version of the current digital twin. What consequences have a change in the physical model onto the behavioral models? What impact has a reduction of maintenance on reliability? The challenge is to build a unified model (the digital twin) from heterogeneous ones (its facets). Multi-model consistency management is still an open problem (see [16] for instance).

I acknowledge any source of information as a model. A script, a program, a configuration file, a diagram, etc. We may explicitly have their metamodel (grammar), or not. They are models anyway. Building a digital twin is gathering a consistent federation of models (connected to a reality) responding to specific evolving usages. In that sense, we can consider digital twin engineering as model engineering, like Tisi et al.

[5]When the RS is a human, this step may be (for the time being) impossible.
[6]possibly a virtual reference system in case of simulation (see VIII).

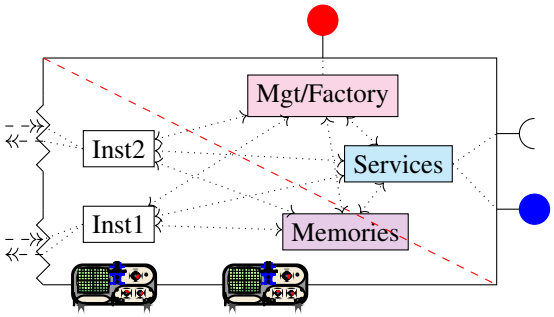

Fig. 6. Insight of digital twin internal with two instances. The red dashed line separates type aspects from instance ones.

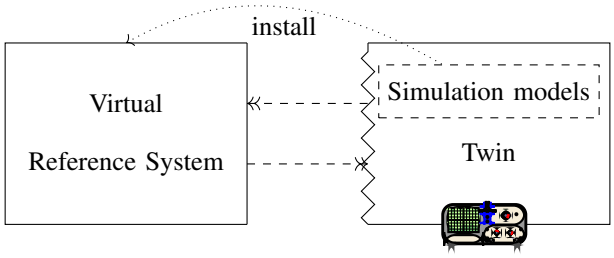

Fig. 7. A digital twin connected to a simulation (a virtual reference system). Installed from simulation models.

suggested in [17]. We think model federation [18] can be a step forward in the capacity of managing interactions among models. This approach helps build new models by linking models and pieces of models together, and assigning a specific semantic to each link.

There are already many types of models. However, I think that specific models for digital twins are yet to be defined. As previously suggested, what realizes the implementation of digital twin is a set of scripts dedicated to implement management and services. For instance the **clone** and **configure** implementations seem critical to me. Here, we probably need new abstractions that could be represented in new models of deployment, configuration, monitoring, cloning, and so on.

Now that we have digital twin as components and their models (or descriptions), we can explore an important difference between type and instances.

VII. Digital twin, type and instances

So far, I have avoided the question of the type/instance difference. This distinction is prevalent in modeling approaches. Since 1994 [19], and more recently in [20], [21], the need for a more complex hierarchy than the usual two levels is acknowledged. For digital twins, an instance is a runnable (or running) digital twin (see figure 4), i.e. a digital twin connected to a reference system. Before this state, a digital twin remains abstract, i.e. configuration values are missing to be fully instantiated. An abstract digital twin defines a digital twin type.

What is the scope defined by a digital twin component? Do we need a single component per (instance of) product? Does a single component ”drive” many copies (instances) of the same product? What about variants of this product? These questions arise the same way in software components. A classical solution is to make a distinction between instances and a *factory* of instances.

Figure 6 refines figure 2. The *memory* stores data of a single instance or many, of its variants (partly configured type), and more generally of types (everything abstract). The *memory* spans the type part and the instance part. The *management* could be a digital twin factory, whose responsibility would be to produce instances of *core* (Inst1 and Inst2 in the figure), the storage for these instances in *memories*, and their *UIs*.

The type level describes information shared by instances. This information can be organize with variants, for instance different possible configurations described in a feature model. This level maps to the conceptual level. The instance level gives all values required to connect to a reference system. It is the running level. As a component, a digital twin, must manage both levels as a unit, consistently.

Now that we have digital twin as components with their models (or descriptions), types and instances, we can explore one important use, beyond mirroring a reference system: simulation.

VIII. Digital twin and simulation

Many digital twin articles refer to simulation, using a digital twin as a simulator of the reference system. I think this is misleading: a digital twin cannot be a simulation. A digital twin requires a reference system to operate. If not a physical one, it could be a simulated one. Then, following figure 7, using a digital twin to collect and analyze data on simulation is possible when the digital twin is connected to a *virtual reference system* (VRS).

The VRS has to provide the expected connection points to the digital twin. If not, the configuration options to a simulation has to generate the proper glue. Nevertheless, special care should be taken with the configuration of digital twins. As an example, in a simulation, time can run very fast (or slow) and GUI may become irrelevant.

Simulation models of the reference system *are* part of the digital twin and are used to install, configure and connect a VRS (figure 7). Data collected by the digital twin through a simulation are also part of the DT. The source of data must be identified, and data analysis, optimization searches, predictions, are again full parts of the digital twin.

Notice that classical simulations - without digital twins - also allow data analysis, optimization searches, and predictions. One difference with a digital twin based simulation is the organization of the simulation. Here again, the paradigm aspect (see II) helps to improve the organization. Real parts of interest have their digital twins, their data and memory, and are organized. Another difference is the ability to mix real systems

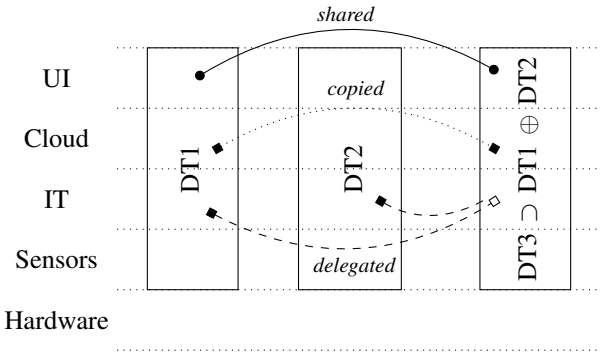

Fig. 8. Digital twins and the reference system cross all layers. Reused parts may be shared, copied or delegated.

and simulations. Since digital twins can be connected to both, it should be possible to build or integrate systems mixing CPS and virtual parts, as we do in software with mockups during test or integration.

IX. Digital twin and digital twins

This last section is no more about digital twins, but about the composition of digital twins. Seen as components (see III), they offer interfaces. Interfaces are great for surface interactions, but what about internal data layers and services like XR or simulations? Figure 8 illustrates the composition problems of digital twins: all systems cross (almost) all layers. I consider 5 layers : hardware, IoT (sensors), IT (business layer), cloud (applications) and user interfaces. This view is simplified. The layers can be physically distributed; IT, cloud and UI rely obviously on hardware and UI are parts of IT and applications. A reference system crosses over all these layers. Whatever its size, a digital twin crosses-over almost all layers; it should have no hardware (but sensors). Simulations have neither hardware, nor sensors (well, not the hardware part of sensors). Hence, composing digital twins requires *composition operators* at each layer. Questions such as "are sensors shared?" or "how logs are merged?" emerge. In fact, many strategies are possible: sharing, delegating (using a proxy), copying, ... and this, at each layer, for each "part". Note that, the reused parts may come from different status installed, configured, or running depending on needs (and probably the composed DT status itself).

My feeling is that in order to be fully operational, digital twin composition cannot but rely on bricks that are digital twin *aware*. Hence, simulators, XR renderers, data storage, etc. have to adopt the digital twin paradigm introduced in section II and offer appropriate services for digital twins.

X. Conclusion

As a paradigm, digital twin is widely used. However, the current digital twin does not exist as a software entity. Articles on digital twins present, with few details, software solutions that are collections of applications, software, data, models with no or very few real connection among them. When a software component model is used, like in [12], nothing specific to digital twins is highlighted. We propose a digital twin architecture close to software components architecture. It would be industrially relevant to define and standardize a management interface based on a digital twin life cycle. Services such as install, configure (to a real environment or to a simulated environment), clone, start, stop, monitor, etc. with high-level parameters (models) are to be considered. The clone operation seems central to me.

The development of a digital twin component produces many kinds of artifacts at different levels of concretization (from instances - everything is configured and valued - to pure abstractions) with possibly many configuration variants. I envision the use of git to manage this diversity (tried, but not evaluated). By chance, git happens to propose a clone operation; this can be of interest. Such code management tools are compatible with iterative development and permit the use of many branches (variants).

I have also argued that the improvement of control through the enhancement of digital twins must trigger the upgrade of the reference system in order to ensure autonomy of the real system. A digital twin cannot operate without a reference system. However, it can be used as an archive or a simulation for offline analysis. On the contrary, a reference system must always be able to be used without its digital twin.

Finally, even if many models are already used for specific aspects of digital twins, I think the most specific models remain to be identified and defined: especially, management and configuration models.

Hence, here are the top challenges I believe we face for making digital twins concrete artifacts:

1) Identifying management models and management services (the component model)
2) Improving capacities of multi-model management
3) Controlling encapsulation in presence of scattered storage for security and privacy
4) Understanding digital twin interactions (including internal interactions) in order to ease their composition
5) Understanding (and tooling) the RS/DT improvement loop

To conclude, digital twin is still at paradigm stage but to become a digital artifact, and a scientific object, a lot of software engineering research has to be performed. Many candidates, none elected.

Acknowledgment

I want to thank Jean-Christophe Bach, Gaëlic Béchu, Caroline Cao, Fabien Dagnat, Salvador Martinez, Quentin Pérez, Christelle Urtado, and Sylvain Vauttier for the fruitful discussions we had.

References

[1] M. Grieves, "Digital twin: manufacturing excellence through virtual factory replication," 2014.

[2] ——, "Product lifecycle management: the new paradigm for enterprises," *International Journal of Product Development*, vol. 2, no. 1-2, pp. 71–84, 2005.

[3] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, "Digital Twin in Industry: State-of-the-Art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, Apr. 2019, conference Name: IEEE Transactions on Industrial Informatics.

[4] R. Minerva, G. M. Lee, and N. Crespi, "Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, Oct. 2020, conference Name: Proceedings of the IEEE.

[5] K. Y. H. Lim, P. Zheng, and C.-H. Chen, "A state-of-the-art survey of Digital Twin: techniques, engineering product lifecycle management and business innovation perspectives," *Journal of Intelligent Manufacturing*, vol. 31, no. 6, pp. 1313–1337, Aug. 2020. [Online]. Available: https://doi.org/10.1007/s10845-019-01512-w

[6] M. Dalibor, N. Jansen, B. Rumpe, D. Schmalzing, L. Wachtmeister, M. Wimmer, and A. Wortmann, "A Cross-Domain Systematic Mapping Study on Software Engineering for Digital Twins," *Journal of Systems and Software*, vol. 193, p. 111361, Nov. 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121222000917

[7] Y. Lin, L. Chen, A. Ali, C. Nugent, C. Ian, R. Li, D. Gao, H. Wang, Y. Wang, and H. Ning, "Human Digital Twin: A Survey," Dec. 2022, arXiv:2212.05937 [cs]. [Online]. Available: http://arxiv.org/abs/2212.05937

[8] H. Boyes and T. Watson, "Digital twins: An analysis framework and open issues," *Computers in Industry*, vol. 143, p. 103763, Dec. 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0166361522001609

[9] A. Walters, "Gemini Principles," Jan. 2019. [Online]. Available: https://www.cdbb.cam.ac.uk/DFTG/GeminiPrinciples

[10] A. Wortmann, "Digital Twin Definitions," https://awortmann.github.io/research/digital_twin_definitions/, 2023.

[11] J. Aldrich, "The power of interoperability: why objects are inevitable," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. Indianapolis Indiana USA: ACM, Oct. 2013, pp. 101–116. [Online]. Available: https://dl.acm.org/doi/10.1145/2509578.2514738

[12] J. Michael and A. Wortmann, "Towards Development Platforms for Digital Twins: A Model-Driven Low-Code Approach," in *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*, A. Dolgui, A. Bernard, D. Lemoine, G. von Cieminski, and D. Romero, Eds. Cham: Springer International Publishing, 2021, vol. 630, pp. 333–341, series Title: IFIP Advances in Information and Communication Technology. [Online]. Available: https://link.springer.com/10.1007/978-3-030-85874-2_35

[13] J. C. Casquina and L. Montecchi, "A proposal for organizing source code variability in the git version control system," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. Leicester United Kingdom: ACM, Sep. 2021, pp. 82–88. [Online]. Available: https://dl.acm.org/doi/10.1145/3461001.3471141

[14] Digital Twin Consortium, "Capabilities Periodic Table." [Online]. Available: https://www.digitaltwinconsortium.org/initiatives/capabilities-periodic-table/

[15] R. Parmar, A. Leiponen, and L. D. W. Thomas, "Building an organizational digital twin," *Business Horizons*, vol. 63, no. 6, pp. 725–736, Nov. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007681320301014

[16] P. Stünkel, H. König, Y. Lamo, and A. Rutle, "Comprehensive Systems: A formal foundation for Multi-Model Consistency Management," *Formal Aspects of Computing*, vol. 33, no. 6, pp. 1067–1114, Dec. 2021. [Online]. Available: https://dl.acm.org/doi/10.1007/s00165-021-00555-2

[17] M. Tisi, H. Bruneliere, J. de Lara, D. Di Ruscio, and D. Kolovos, "Towards Twin-Driven Engineering: Overview of the State-of-The-Art and Research Directions," in *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*, ser. IFIP Advances in Information and Communication Technology, A. Dolgui, A. Bernard, D. Lemoine, G. von Cieminski, and D. Romero, Eds. Cham: Springer International Publishing, 2021, pp. 351–359.

[18] C. Guychard, S. Guérin, A. Koudri, F. Dagnat, and A. Beugnard, "Conceptual interoperability through Models Federation," in *Semantic Information Federation Community Workshop*, 2013.

[19] Jim Odell, "Power Types," *J. Object-Oriented Programming, 7(2)*, pp. 8–12, 1994.

[20] C. Atkinson and T. Kühne, "The Essence of Multilevel Metamodeling," in *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, G. Goos, J. Hartmanis, J. van Leeuwen, M. Gogolla, and C. Kobryn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 2185, pp. 19–33, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-45441-1_3

[21] C. Atkinson, M. Gutheil, and B. Kennel, "A Flexible Infrastructure for Multilevel Language Engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 742–755, Nov. 2009, conference Name: IEEE Transactions on Software Engineering.