Towards Flexible Evolution of Digital Twins with Fluent APIs

Daniel Lehner*†, Antonio Garmendia†, Manuel Wimmer*†
* Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT)
† Institute for Business Informatics - Software Engineering
Johannes Kepler University Linz, Science Park 3, 4020 Linz, Austria
{firstname}.{lastname}@jku.at
https://se.jku.at

***Abstract*—With the increase of technologies such as the Internet of Things (IoT) and Cyber-Physical Systems, a huge amount of data is generated by current systems. To gain insights from this data, it must be combined with meta-information about its origins. Therefore, Digital Twins (DTs), as a common representation of a system and its data, are currently gaining traction in both industry and academia. However, these DTs have of course to be evolvable in order to reflect the high need of flexibility of the systems to support extensions, adaptations, customizations, etc. Evolving the DT representations currently not only involves a lot of manual effort, but might also lead to loss of data if not done correctly. To provide dedicated evolution support, we propose a dedicated framework for realizing evolution strategies between the schema, instance, and data level of a DT. In particular, we present a fluent API which allows the flexible but systematic manipulation of DTs during runtime and demonstrate its usage for a use case.**

***Index Terms*—Digital Twin, Evolution, Maintenance, Fluent APIs**

I. Introduction

Nowadays, Cyber-Physical Systems (CPSs) are becoming a common practice to control physical processes [1]. Specifically, these processes are often real-time monitored, for which a large amount of data is generated.

To make sense out of this data, it must be combined with meta-information about the physical system [2]. Therefore, Digital Twins (DTs) have emerged over the last years as a virtual representation of such systems that allow a bi-directional data flow [3]. Several services can make use of this virtual representation to provide functionality such as prediction, simulation, visualization, or system control [4], [5]. Therefore, a DT usually comprises the current snapshot of a system, that is further defined by a schema adding the meta-information perspective. Additionally, historical data can be stored by the DT for time-series analysis, prediction, or simulation. Some dedicated support in terms of so-called DT platforms (e.g., Azure[1], Eclipse[2], or AWS[3]) has been proposed already. Whereas these platforms usually focus on the schema and snapshot level, they can make use of a so-called Time-Series Database (TSDB) to store historical data. Some of these TSDBs even provide support for integrating schemas (e.g., Time-Series Insights service by Microsoft[4], or the Timescale database[5]). However, the schema of a DT is subject to frequent change in design, because the evolution of the physical system should also be represented by its DT [6]. A change in the physical system may trigger an evolution step in the DT. However, as there are several levels of a DT (i.e., schema, snapshot, historical data), such an evolution step might trigger several co-evolution steps at different levels, as shown in Fig. 1. As the `Historical Data` of a DT conforms to the structure imposed by the `Snapshot`, an evolution of this Snapshot requires co-evolution adaptations on the Historical Data to ensure consistency. As the Snapshot must conform to the `Schema` of a DT, evolution of this schema also triggers respective co-evolution steps on the Snapshot level.

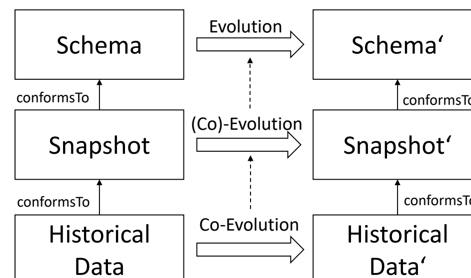

Fig. 1. Overview of evolution levels in a DT.

In the literature, there already exist co-evolution solutions for models [7]–[9] and relational databases [10]–[12] (e.g. convert one data type to another). The evolution of DTs has an added complexity, which is that the logical models (schemas and snapshots) and the TSDB must be evolved synchronously [2].

The current digital twin platforms do not provide dedicated support for evolution scenarios regarding a DT, especially when the schema is changed and its existing data has to co-evolve. Consequently, (i) a lot of manual effort for synchronizing adaptations in the physical system with its virtual representation is required, (ii) the evolution of the schema must ensure the compatibility with the current applications,

[1] https://azure.microsoft.com/services/digital-twins/
[2] https://www.eclipse.org/ditto/
[3] https://aws.amazon.com/greengrass/
[4] https://azure.microsoft.com/services/time-series-insights/
[5] https://www.timescale.com/

and (iii) unintended data loss may occur, if this synchronization is not performed correctly. To avoid this, the maintenance engineer should perform a set of decisions to manage a systematic maintenance process. Therefore, in this paper, we propose a fluent API[6] for handling such evolution scenarios for DTs. We propose a framework that can be integrated into existing DT architectures only requiring a base interface to manipulate digital twin elements. This framework provides a fluent API to support different decisions during maintenance of a DT. These decisions include migration, versioning, or completely dropping elements as they are no longer required. To examplify our approach, we provide a first demonstration use case.

The remainder of this paper is structured as follows. Section II describes a running example of an air quality control use case, while Section III outlines our approach and shows its application for the presented use case. Finally, Section IV concludes with an outlook on future work.

II. Motivation

To motivate our work, we consider an air quality use case based on the description in [13]. In this use case, the CO_2 values for individual rooms in a building are measured. Therefore, in each room, a CO_2 sensor is connected to a controller that regularly sends CO_2 values to its DT. This DT runs as a service in the cloud which is a virtual representation of the schema, snapshot and historical data of the physical system that can be consumed by different services.

A. Example DT Language

As described above, there exist several languages to describe DTs for a system. In Fig. 2, we introduce an example DT language that is used in the remainder of this paper. In this language, the schema of a system is represented by `Types` (e.g., Room, Controller, Sensor) and `Properties` (e.g., co2Value of a Sensor). After doing adaptations, old versions of these Types and Properties can be casted to VersionedTypes and VersionedProperties, respectively, to indicate that they do no longer belong to the most current schema of the system.

The system snapshot is described using `Instances` (e.g., Room101, Raspberry1, the DHT811 CO_2 Sensor connected to Raspberry1 in Room101) and `Slots` that cover specific data entries. The current `Entry` is part of the snapshot, whereas historical entries can be persisted as historical data. To distinguish between different entries for the same slot, the timestamp is stored as well.

B. Evolution Case

After a DT is implemented using a language as described above, there might be adaptations in the physical system that have to be reflected by the DT. To keep the DT definition consistent with the actual system, these changes must be propagated between all levels of the DT. First, the schema might be adapted. Based on this schema adaptation, there might be changes to the instances that already conform to

[6]https://martinfowler.com/bliki/FluentInterface.html

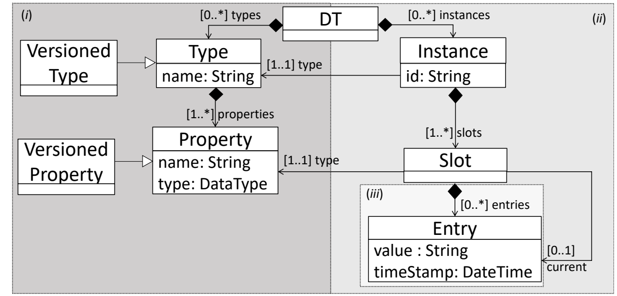

Fig. 2. Metamodel for representing DT with (i) schemas, (ii) snapshots, and (iii) historical data.

the previous version of the schema. Deriving these changes can already be covered using existing approaches. However, in a DT language, these changes in the schema and snapshot level must also be reflected in the historical data that is already stored in a DT. Usually, there are several options of what to do with this data, that have to be considered by the person performing the migration. To make this more tangible, we present an example in the following.

Example: Change property name for measured CO_2 values (cf. Fig. 3): One of such changes might be that after replacing an old sensor with a new version, the CO_2 values are reported using the name $\texttt{co}_2$ instead of $\texttt{co}_2\texttt{Value}$ (i.e., $\texttt{co}_2\texttt{Value}$ must be changed to $\texttt{co}_2$). Therefore, in the existing DT schema, (i) a new Property called $\texttt{co}_2$ is added, and (ii) the existing property $\texttt{co}_2\texttt{Value}$ is casted to `VersionedProperty`. This change can be automatically propagated to the snapshot level by creating a slot named co_2 for each instance that conforms to CO2Sensor. However, after adapting the schema and snapshot, the existing CO_2 values that were already measured for these existing instances must be adapted as well. As there are several alternatives how to perform this step, this cannot be done automatically. An expert must decide between the following options: (1) `Migrate` instances to new schema. Using this option, both historical data that was measured before the migration, and newly measured data is available via the Property $\texttt{co}_2$. In this option, copying the data may require a lot of computational effort. But, the co_2Value property can be deleted afterwards. (2) `Version` the previous element in addition to introducing the new one. One of these instances still conforms to the old schema, and data before the versioning is available via the $\texttt{co}_2\texttt{Value}$ property of this instance. The other instance conforms to the new schema and data measured after versioning is available via the co_2 property from this instance. This option does not require any copy effort, but requires to keep the old Property besides the new one. (3) `Drop` historical data by deleting the co_2Value Property. This option does not require any copy operations as the data is simply forgotten.

III. Approach

In this section, we present the architecture as well as the service for evolving DTs and show its application for the previously introduced evolution case.

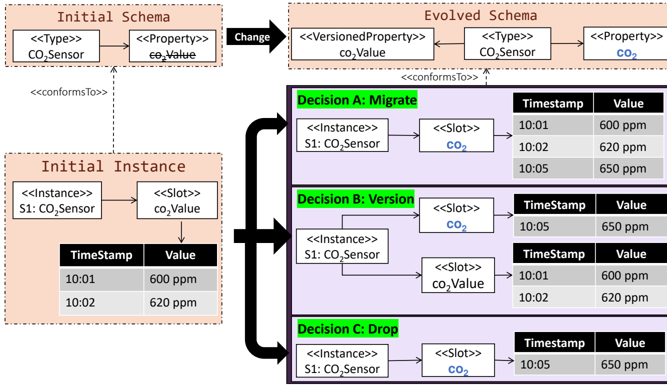

Fig. 3. Example of a change property co-evolution.

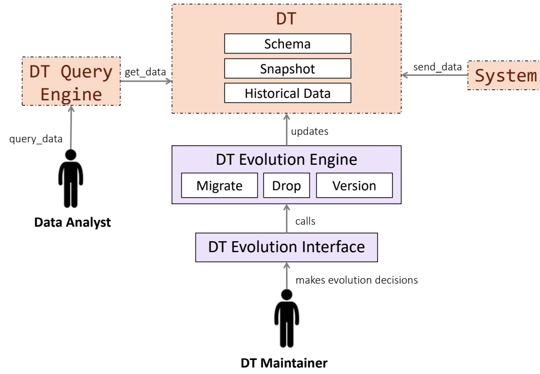

Fig. 4. Proposed Co-Evolution Architecture for DTs.

TABLE I
DESCRIPTION OF EVOLUTION AND CO-EVOLUTION OPERATIONS FOR THE KERNEL DT META-MODEL.

Schema Evolution		Snapshot Co-Evolution	
Operation	**Elements**	**Operation**	**Elements**
Create	Type	/	/
Create	Property	Create	Slot
Update	Type.name	Create	Instance
Update	Property.name	Create	Slot
Update	Property.type	Create	Slot
Delete	Type	Delete	Instance
Delete	Property	Delete	Slot

A. Co-Evolution Architecture

In previous work, we have developed an architecture for leveraging model repositories for DTs [14]. Inspired by this work, we designed the following architecture for managing co-evolutions of DTs that is shown in Fig. 4. In this architecture, the `DT` can receive data from the physical system during its runtime. This data must conform to the schema imposed by the DT, and is represented by the DT as the current snapshot. After an update is sent, the previous snapshot is saved as historical data. A `Data Analyst` can query this stored snapshot and historical data using a `DT Query Engine` to gain insights from the operation of the running system. Changes to the schema of a system are performed by a `DT Maintainer`. By adapting the schema, the current snapshot is migrated automatically. For the historical data, a DT Maintainer must explicitly define whether it should be dropped, versioned, or maintained, by describing the decision using the `DT Evolution Engine`. This DT Evolution Engine is described in more details in Section III-B. After the DT Evolution Engine performs the co-evolution of the system based on this description, the information can be again retrieved in the expected manner, e.g., by the Data Analyst.

B. DT Evolution Service

Every change in the physical system must also be reflected in the DT. As there are different levels of a DT (i.e., schema, snapshot, and historical data), each evolution might trigger several co-evolution operation. To describe possible evolutions of the schema, evolution operations can be derived from the general DT language (e.g., as shown by Hermannsdörfer et al. [15] or Berardinelli et al. [9]). Depending on the performed evolution operation on the schema level, co-evolution operations on the snapshot level can be triggered automatically to ensure consistency. These operations are shown in Table I for the DT language depicted in Figure 2.

Creating a new Property, or updating the name or schema of an existing Property, requires to add a new Slot for this property on the snapshot side. In the update cases, the old Property should be casted to VersionedProperty afterwards. Updating the name of a Type requires to add a new Instance for each existing instance that conforms to this new Type. The old Type should afterwards be casted to VersionedType.

However, for every change on the snapshot level, historical data for specific instances must be considered as well. Therefore, different options are possible, as also shown in Figure 3:

1) Migrating historical data requires additional computational effort for copying entries from one slot to another, but then, the versioned information on the schema and instance level can be deleted.
2) Versioning does not require any copy operation for historical data. Therefore, the old version of the schema must be maintained using VersionedTypes or VersionedProperties.
3) Dropping historical values also does not require any copy operation, and additionally allows to delete versioned information on the schema level. However, using this option, historical data is lost.

All required operations must be implemented in the respective elements of the DT, based on the used language. For each change in the physical system, a DT maintainer can then specify (i) required changes on the schema level, and (ii) what to do with historical data. To guide the DT maintainer through this process, the DT Evolution service should provide an interface for entering decisions that lead to a co-evolution strategy. Such an interface can be e.g., implemented using a RESTful interface [16], a fluent API, a command-line interface [17], or even a chatbot [18].

Listing 1. Evolution Operations for Types

```
1 context Type op migrateData(src String, trg String):
2     foreach instance in self.allInstances(){
3         data <- instance.getEntriesForSlot(src);
4         instance.createEntries(trg, data);
5     }
6     self.deleteProperty(src)
7
8 context Type op dropProperty(prop String):
9     self.deleteProperty(prop);
10
11 context Type op createProperty(name String):
12     self.getProperties().add(new Property(name))
13     foreach instance in self.allInstances(){
14         instance.addSlot(name);
15     }
16
17 context Type op deleteProperty(prop String):
18     self.getProperties().remove(prop)
19     foreach instance in self.allInstances(){
20         instance.deleteSlot(prop);
21     }
```

Listing 2. Formulating Decisions using the Fluent Evolution Interface

```
1 -- version decision
2 CO2Sensor.createProperty('co2')
      ↪ .versionProperty('co2Value');
3
4 -- migration decision
5 CO2Sensor.createProperty('co2')
      ↪ .versionProperty('co2Value')
      ↪ .migrateData('co2Value', 'co2');
6
7 -- drop decision
8 CO2Sensor.createProperty('co2')
      ↪ .versionProperty('co2Value')
      ↪ .dropProperty('co2Value');
```

C. Demonstration

We now demonstrate the DT evolution service as it can be used by a DT maintainer to describe co-evolution decisions based on the afore presented example evolution case. We use a fluent API approach to describe the three decision types (migrate, version, drop) in List. 2.

List. 1 shows the evolution operations which are used by List. 2. The schema evolution is described in List. 2. In all options, the new property is created via the operation $createProperty('co2')$, and the predecessor property is casted to a VersionedProperty via the operation $versionProperty('co2Value')$.

As snapshot co-evolution, a new slot called $'co2'$ is automatically added to each instance which is instantiated from the CO2Sensor type (cf. List. 1, lines 11–15). For versioning of the historical data, no more additional logic is required.

For the drop option (cf. List. 2, line 8), the $co2Value$ VersionedProperty is simply deleted within the operation $self.dropProperty('co2Value')$. With the $deleteProperty$ operation, all existing slots (and thus corresponding historical data) are deleted as well (cf. List. 1, lines 17–21).

For the migration option (cf. Listing 2, line 4), the operation $migrateData('co2Value','co2')$ is called. This operation (cf. Listing 1, lines 1–6) reads all historical data from the VersionedProperty $co2Value$, and re-creates the data for the new $co2$ Property. After this migration is finished, the $co2Value$ VersionedProperty is deleted (in the same way as for the drop option) as it is no longer needed.

IV. Conclusion & Next Steps

In this paper, we introduced a fluent API to manage the evolution of the schema, snapshot, and historical data of a DT. In the future, we plan to realize the envisioned DT Evolution Engine for different DT platforms. By conducting experiments using several case studies, we aim to explore trade-offs between the presented evolution strategies as well as additional ones.

Acknowledgements

Work partially funded by the Austrian Science Fund (P 30525-N31) and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).

References

[1] S. Biffl, A. Lüder, and D. Gerhard, *Multi-Disciplinary Engineering for Cyber-Physical Production Systems*. Springer, 2017.

[2] A. Mazak, S. Wolny, A. Gómez, J. Cabot, M. Wimmer, and G. Kappel, "Temporal models on time series databases," *JOT*, vol. 19, no. 3, pp. 3:1–15, 2020.

[3] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[4] D. Jones, C. Snider, A. Nassehi, J. Yon, and B. Hicks, "Characterising the digital twin: A systematic literature review," *CIRP-JMST*, vol. 29, pp. 36–52, 2020.

[5] V. Kuliaev, U. D. Atmojo, S. Sierla, J. O. Blech, and V. Vyatkin, "Towards product centric manufacturing: From digital twins to product assembly," in *Proc. of INDIN*, pp. 164–171, IEEE, 2019.

[6] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[7] M. Herrmannsdörfer and G. Wachsmuth, "Coupled evolution of software metamodels and models," in *Evolving Software Systems*, pp. 33–63, Springer, 2014.

[8] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *TSE*, vol. 43, no. 5, pp. 396–414, 2016.

[9] L. Berardinelli, R. Drath, E. Maetzler, and M. Wimmer, "On the evolution of CAEX: A language engineering perspective," in *Proc. of ETFA*, pp. 1–8, IEEE, 2016.

[10] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner, "Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language," in *Proc. of SIGMOD*, pp. 1101–1116, 2017.

[11] E. Domínguez, J. Lloret, Á. L. Rubio, and M. A. Zapata, "Medea: A database evolution architecture with traceability," *Data & Knowledge Engineering*, vol. 65, no. 3, pp. 419–441, 2008.

[12] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the prism workbench," in *Proc. of VLDB*, pp. 761–772, 2008.

[13] D. Lehner, S. Wolny, M. Vierhauser, W. Narzt, and M. Wimmer, "AML4DT: A Model-Driven Framework forDeveloping and Maintaining Digital Twins with AutomationML," in *Proc. of ETFA*, IEEE, 2021.

[14] D. Lehner, S. Wolny, A. Mazak-Huemer, and M. Wimmer, "Towards a reference architecture for leveraging model repositories for digital twins," in *Proc. of ETFA*, pp. 1077–1080, IEEE, 2020.

[15] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," in *Proc. of SLE*, pp. 163–182, Springer, 2010.

[16] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly", 2008.

[17] B. D. Davison and H. Hirsh, "Toward an adaptive command line interface," in *Proc. of HCI (2)*, pp. 505–508, 1997.

[18] H. Ed-Douibi, G. Daniel, and J. Cabot, "OpenAPI Bot: A Chatbot to Help You Understand REST APIs," in *Proc. of ICWE*, pp. 538–542, Springer, 2020.