A consistency management framework for Digital Twin models

Hossain Muhammad Muctadir[a], Eduard Kamburjan[c,d], Loek Cleophas[a,b], Mark van den Brand[a]

[a] *Software Engineering and Technology cluster, Eindhoven University of Technology, Eindhoven, The Netherlands*
[b] *Computer Science division, Stellenbosch University, Stellenbosch, South Africa*
[c] *IT University of Copenhagen, Copenhagen, Denmark*
[d] *University of Oslo, Oslo, Norway*

Abstract

Digital twins (DTs) encapsulate the concept of a real-world entity (RE) and corresponding bidirectionally connected virtual one (VE) mimicking certain aspects of the former in order to facilitate various use-cases such as predictive maintenance. DTs typically encompass various models that are often developed by experts from different domains using diverse tools. To maintain consistency among these models and ensure the continued functioning of the system, effective identification of any consistency issues and addressing them whenever necessary is imperative. In this paper, we investigate the concept of consistency management and propose a consistency management framework that addresses various characteristics of DT models. Subsequently, we present three working examples that implement the proposed framework with graph-based techniques. Taking the working examples into account, we demonstrate and argue that our consistency management framework can provide crucial assistance in the consistency management of DT models.

Keywords: digital twin, consistency management, inconsistency detection, consistency management framework

⋆This research was funded by NWO (the Dutch national research council) under the NWO AES Perspectief program, project code P18-03 P3. We acknowledge the contributions of Judith Houdijk and Mohammad Ibrahim to the working examples.

Email addresses: h.m.muctadir@tue.nl (Hossain Muhammad Muctadir), eduard.kamburjan@itu.dk (Eduard Kamburjan), l.g.w.a.cleophas@tue.nl (Loek Cleophas), m.g.j.v.d.brand@tue.nl (Mark van den Brand)

1. Introduction

A digital twin (DT) is a software-intensive system in which certain aspects of a real-world entity (RE) are mimicked in one or more connected virtual entities (VE) in order to facilitate various advanced use-cases such as predictive maintenance, optimization, virtual training, etc. Recently, the concept of DTs has been integrated and used in an increasing number of domains [1]. A significant number of these DTs are related to complex industrial systems where various heterogeneous and cross-domain models and model-driven engineering (MDE) play a central role [2, 3, 4, 5]. VEs abstract real-world concepts, similar to models [6], making them inherently model-based [7]. Consequently, the benefits and challenges of MDE also apply to VEs. In this paper, we focus on model consistency management, which is one of the well-known challenges in MDE, but now in the context of VEs of DTs. We see system-level consistency as an aggregation of model-level consistency, managing which is the focus of our research.

Maintaining consistency among models within a DT is crucial and challenging [8]. This is mainly due to the heterogeneous nature of the DT models, which is the result of using various commercial and in-house tools in the design and development of these models where domain-specific languages (DSL) play an increasingly significant role [9, 10, 7]. Despite various standardization efforts, this continues to be a challenge [7]. The frequent evolution of these models only compounds the situation, as it introduces additional complexity [8, 11]. We identify these as key challenges for seamless consistency analysis. With an interview study with DT practitioners [7], we discovered that consistency problems frequently hinder the design, development, and maintenance processes of DT models. We also found that the existing consistency management and detection methods are not utilized in addressing these issues. Our analysis of the existing consistency management research revealed that there are no methods specifically developed for DT models. Furthermore, the majority of these methods are either highly theoretical or technology- or tool-specific, which impedes their practical application to complex DT systems. Our work aims to provide a standard way to address this consistency management challenge, considering the heterogeneous and complex nature of DT models.

To establish a method for consistency management of DT models, it is es-

sential to first analyze their distinguishing features compared to conventional MDE models and evaluate the implications these differences have for consistency management. Since the concept of DT and its extensive adoption are rather recent, it remains unclear whether DT models exhibit any distinguishing characteristics. Recent studies [12, 13, 11] and our interaction with DT practitioners [7] provided us with early indications about the particularities of DT models, especially those used in developing VE. We found that these models are highly multi-domain, heterogeneous, and evolve rather frequently while maintaining sufficient accuracy, trustworthiness, efficiency, and speed when executed. Moreover, we observe that these models, intended to be used collectively, are frequently developed by engineers from different disciplines using a variety of tools. Despite sharing some characteristics with traditional Model Based Systems (MBSs), DT models are potentially more complex to manage due to factors such as higher heterogeneity, frequent evolution, and their multi-domain multi-tool nature. Furthermore, managing VE models that are reused or repurposed from their corresponding RE can become even more complex, as they operate beyond their original context; this additional complexity also holds for models originating from or involving AI algorithms.

In this paper, we present a conceptual framework for managing the consistency of DT models. To develop this framework, we began by defining the concept of consistency management to establish a common understanding. We then performed a thorough analysis of various known characteristics inherent to DT models. This analysis allowed us to elicit a set of requirements that we claim are essential for any system aiming to manage the consistency of these specific and complex models. Afterwards, the framework was developed with these DT-derived characteristics explicitly in mind. We evaluate the framework based on three different working examples where it was implemented, demonstrating its capabilities, effectiveness, applicability, and flexibility. Finally, we discuss its limitations and possibilities for future improvement.

The paper is structured as follows. In Section 2, we explore relevant research to manage the consistency of models. The definition of consistency management and the characteristics of the DT models are discussed in Sections 3 and 4, respectively. We introduce our consistency management framework in Section 5. The three working examples implementing this framework are presented in Section 8. The evaluation of the framework based on the results from the working examples is presented in Section 9. The threats to the validity of our approach are discussed in Section 11. In Section 12, we

discuss further research possibilities for extending and improving our consistency management framework. Finally, Section 13 concludes the paper.

This paper is an extension of our conference paper accepted at the 50th Euromicro conference on Software Engineering and Advanced Applications (SEAA) [14]. The following additions have been made:

- The related work presented in Section 2 has been significantly extended.
- The definition of consistency management in Section 3 has been extended with the notion of consistency rule management.
- We re-evaluated the literature analysis in Section 4.1 adding three papers in Table 1.
- The consistency management framework at Section 5 has been significantly extended with classes that facilitate the management of consistency rules, explained in Section 5.4.
- We included a implementation template in Section 6 for implementing the framework.
- In Section 8, the previous two working examples (cf. Section 8.1) have been extended in relation to the extension of the framework. We also improved the overall discussion with more examples. In Section 8.2 we added a third working example implementing consistency management within the robotic domain.
- The discussion on the evaluation of the framework in Section 9 has been extended based on the extension of the consistency management framework and the updated working examples.
- In Section 10, we included a discussion on practical lessons learned from implementing the working examples.
- We added Section 11 discussing the threats to the validity of our approach.
- A new Section 12 discusses the possibilities for extending this research.

2. Related work

As indicated earlier, DTs are complex systems encapsulating intricate interplay among various software and hardware components, which can be prone to continuous evolution. The development and maintenance of such systems require constant collaboration among several engineering domains. In an environment as dynamic as this, the emergence of consistency issues is highly probable. Inconsistency detection and management is a well-researched topic that extends beyond DTs. In this section, we explore the existing literature on this topic and assess their applicability in the context of DTs.

A large number of DT definitions concur that within a DT, certain aspects of the RE should be virtually mimicked by the corresponding VE [1]. Over time, one entity's behavioral or structural characteristics may diverge more significantly from another due to factors like upgrades or wear from consistent use. Identifying and addressing these drifts in time is crucial for the effective functioning of the corresponding DT. In recent years, researchers have focused on this topic. For instance, Seok et al. [15] developed a method for maintaining the fidelity of the VE by sensor data-based iterative synchronization of the VE with its physical counterpart. Kamburjan et al. [16] analyze the structure of a VE to detect whether it is indeed mirroring its physical counterpart and, thus, can be used as its meaningful representation.

Managing cross-domain models, which form the basis of a DT, is a fundamental activity in a DT's development and maintenance. According to David et al. [17], a consistent system is not necessarily correct, yet it is a significant heuristic for achieving eventual correctness. Inspired by database management systems, Dolk and Konsynski [18] defined model management as the activity of storing, manipulating, and retrieving models. However, in the context of DTs, model management is complicated by the models having certain specialized characteristics, such as higher heterogeneity, frequent evolution, and their multi-domain multi-tool nature. As these models evolve, they must be kept consistent with each other for the proper functioning of the DT. Any inconsistency should be identified automatically and immediately reported to the engineers. Furthermore, managing VE models that are reused or repurposed from their corresponding RE can become even more complex, as they operate beyond their original context; this additional complexity also holds for models originating from or involving machine learning (ML) algorithms. All these factors suggest the necessity of an appropriate

model management system that allows storing dependency relations over the heterogeneous cross-domain models, automatically updating these relations as models are modified, and detecting consistency issues based on the stored information.

Inconsistency detection among software models, design artifacts, and related entities is a well-established research topic – for example, Balzer [19] discusses consistency as early as 1989, and, among others, Pascual et al. [20, 21] discuss the difficulties of properly defining consistency in this setting. Feldmann et al. [22] studied the state of the art in consistency management approaches and classified them into *proof-theory-based*, *synchronization-based*, and *rule-based*.

The *proof-theory-based* approaches [23, 24, 25] are suitable for well-defined formal systems where logically correct conclusions can be drawn based on various theorems and axioms. The primary limitation of these methods is the need to convert diverse multi-domain models, typically semi-formal or informal, into complex formal representations.

With *synchronization-based* approaches, transformations are devised to specify how statements in one model correspond to statements in a connected model. Using these transformation rules, models are synchronized and checked for consistency [22]. Giese et al. [26] implemented the model synchronization through triple graph grammers. This approach requires a separate definition of every relation between model elements, making it highly resource-intensive. Gausemeier et al. [27] addressed this limitation using cross-domain system specification, allowing the engineers of the involved domains a common understanding of the system. Although several other *synchronization-based* approaches have been proposed [28, 29], they are often challenging to implement within heterogeneous modeling environments. This is due to the increasing number and complexity of transformation rules with growing model relations [22]. Our method, presented in Section 5, takes the model heterogeneity into account and contains mechanisms to address this in a unified manner.

With *rule-based* approaches, rules can be described as conditions indicating consistent relations among relevant models or indicating inconsistency. These are used more frequently for consistency management than the other two approaches. These methods are more adaptable for cases where knowledge of the underlying system is incomplete [22]. There are a number of publications proposing different *rule-based* approaches for consistency detection and resolution [30, 31, 32, 33, 34]. We discuss the prominent ones in the

following.

Mens et al. [30, 35] proposed a graph transformation-based model inconsistency detection and resolution method where the inconsistencies are represented as transformation rules. They used critical pair analysis, a method typically used in graph rewriting to identify minimal examples of a conflicting situation, to identify inconsistencies.

Feldmann et al. published a series of papers focusing on *rule-based* intermodel consistency management [31, 36, 33, 37]. In [37] the authors proposed a conceptual approach for inconsistency detection and performed a working example demonstrating its technical viability. They used semantic web technologies for knowledge representation and querying encoded inconsistencies. This work was further extended to include different levels of graph-based visualization of models and detected inconsistencies, where lower-level visuals offer more information to pinpoint the latter [33]. A formal definition of the inconsistency management problem is presented in [36] where they define the concept of links, which establish relationships among cross-domain models, and assessable consistency relations. This work was extended with DSLs for modeling links and pattern-based consistency rules [31].

There exists a significant body of research that addresses consistency maintenance and management for specific domains and technologies. [38] et al. presented an approach for maintaining consistency between SysML-based system architecture and Modelica-based dynamic models for mechatronic systems. [39] et al. developed a rule-based approach and applied it to detect structural inconsistencies between UML and requirement models.

All these approaches demonstrated promising results for the respective use-cases, domains, and technological spaces. Some of the approaches utilize highly theoretical concepts, which often have a steep learning curve, making their practical implementation more challenging. We also identified methods that were developed for certain types of models or require significant manual intervention, which we deem unsuitable for continuously evolving DT models. Modern integrated modeling tools (i.e., Simulink) support some inconsistency checking for models developed with them. Yet Torres et al. [40] note that most of these tools lack proper support for consistency checking for models developed in different tools. Furthermore, we were unable to find any approach tailored for the consistency management of DT models. Therefore, a model management tool capable of maintaining consistencies among cross-domain models is required to deal with the DT model consistency challenge.

3. Model consistency and its management

Consistency management is of utmost importance for handling long-running heterogeneous systems, as inconsistencies between models will eventually result in incorrect behaviour of such systems. We require a precise definition of the notion of consistency itself to derive requirements for a *practical* management system. In this section, we discuss our conceptualization of consistency management. Our focus is on capturing the relationships among models in an MBS. These relations are used to identify potential violations of these relations and to record the information necessary to address them. The concrete nature of subsequent actions, such as confirming identified issues by investigating the MBS and resolving them, depend on the exact implementation and may require a human in the loop [7].

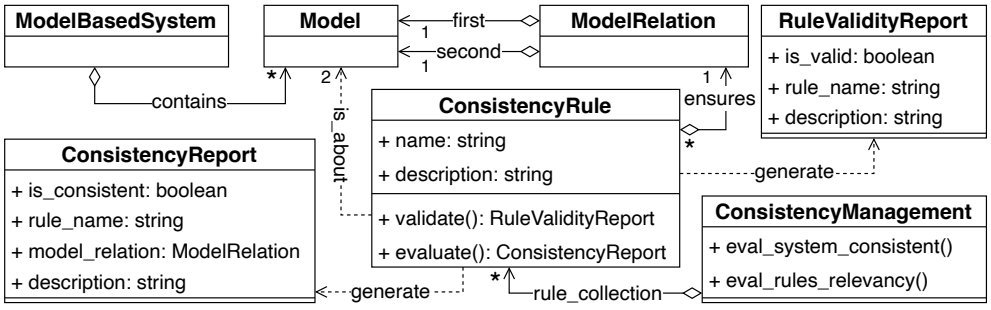


Figure 1: Meta-model showing the concepts related to consistency management

Figure 1 shows the meta-model that underlies our definition of model consistency management. We consider consistency management as the *process of maintaining consistency among interrelated models, according to a set of rules that define consistency* within a DT. The core activities of this process are as follows:

(A1) Rule-based consistency maintenance among models used in an MBS, i.e., the application of consistency rules to detect violations.

(A2) The management of the consistency rules themselves.

Let us first look at the basic notions: MBS, model, and model relation. Within a `ModelBasedSystem`, various relationships can be established between any two `Models`. The actual nature and characteristics of this relationship depend on the implementation of the corresponding models and on

the use-case. In our meta-model, which is shown in Figure 1, a relationship is expressed with the concept `ModelRelation`, which establishes a relationship between two models.

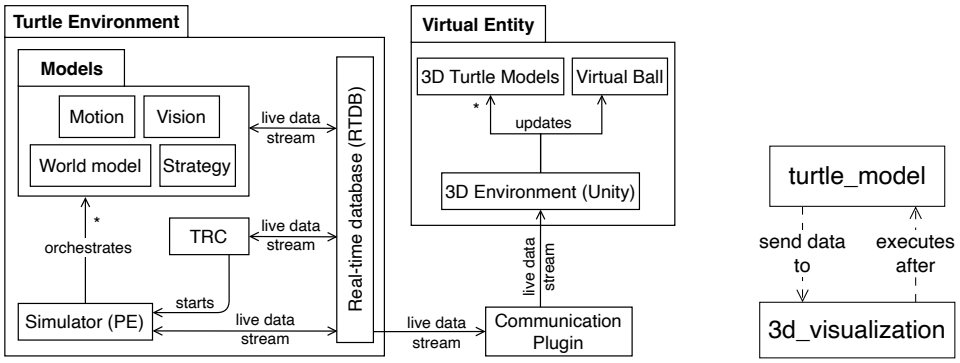
The central notion for consistency management is the `ConsistencyRule`. One or more `ConsistencyRule`s ensure the validity of a `ModelRelation`, which corresponds to the activity **A1**. This is performed using the `evaluate` function, which generates a `ConsistencyReport` containing the details of the corresponding `ModelRelation`, `ConsistencyRule`, and any potential consistency issue with the boolean flag `is_consistent`. Additionally, the `description` property provides a comprehensible explanation of the aforementioned details. An MBS may encounter consistency issues if any of the consistency rules are violated (i.e., `is_consistent` is set to `false`).

The second activity, **A2**, i.e., managing consistency rules, is about maintaining the relevancy and validity of the `ConsistencyRule`s. This is needed as MBS and the `Model`s contained within evolve over time [4, 41, 42], thus changing the nature of the corresponding existing `ModelRelation`s, e.g., by removing some of them or adding new ones. Such evolution can render some of the existing `ConsistencyRule`s invalid. This is addressed by periodically analyzing the `ConsistencyRule`s for their relevancy and validity, generating `RuleValidityReport`s, each of which indicates the validity status of a specific rule with the boolean property `is_valid` along with the name of the rule and a short description for traceability.

Finally, the concept of `ConsistencyManagement` is the aggregated evaluation of all `ConsistencyRule`s existing within an MBS, analyzing their validity, and, if necessary, acting on the generated reports (i.e., `ConsistencyReport` and `RuleValidityReport`).

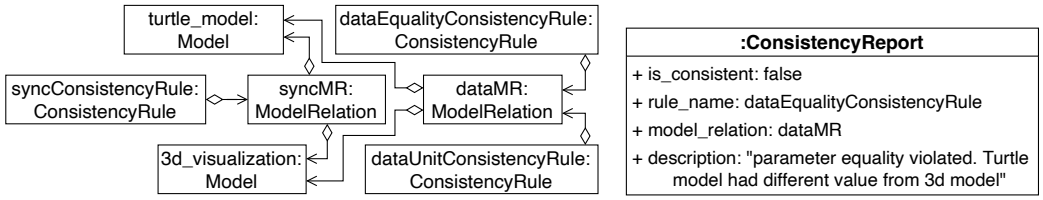
To understand the need for explicit consistency management, consider the example in Figure 2, which is based on a digital shadow for autonomous soccer robots, called Turtles, developed by Walravens et al. [13]. As depicted in Figure 2a, data is generated by the software models, which are responsible for the behavior of the physical Turtle robots, and are visualized in a 3D environment using virtual models of the robots. Figure 2b is a simplified representation of the relation between the software and 3D models denoted as `turtle_model` and `3d_visualization` respectively.

Figure 3a presents the entities needed for the consistency management of these models. They are related in two ways: `3d_visualization` (1) receives data from and (2) starts execution (i.e., visualization of the received data) after `turtle_model`. These model relations are represented in Fig-



(a) (b)

Figure 2: The architecture of the Turtle digital shadow. Figure (a) shows the detailed architecture from [13] and (b) shows a simplified representation focusing on the relation between software models responsible for the behavior of the Turtle soccer robots and 3D visualization model. The latter is used in Figure 3.



(a) (b)

Figure 3: Consistency management for Turtle digital shadow shown in Figure 2. Figure (a) shows instances of entities needed for consistency management of models and their relations depicted in Figure 2b and (b) an example consistency report generated by `dataEqualityConsistencyRule`.

ure 3a with `dataMR` and `syncMR` respectively, which are instances of `ModelRelation`. The consistency of these relations is maintained by three instances of `ConsistencyRule`: `syncConsistencyRule`, `dataUnitConsistencyRule`, and `dataEqualityConsistencyRule`, where the former is related to `syncMR` and the latter two to `dataMR`. The `syncConsistencyRule` ensures the synchronization between `turtle_model` and `3d_visualization` where the latter must execute after receiving data from the former. The other two rules ensure the consistency of the data exchanged between the corresponding models. The first one ensures the data is in the correct unit, while the second one verifies the equality of the seed value of both models. In this example, consistency management entails the periodic evaluation of the

three `ConsistencyRule`s mentioned for identifying any consistency issues, the analysis of these rules to verify their validity and integrity, and promptly reporting any violations. For example, Figure 3b shows a consistency report resulting from a violation of `dataEqualityConsistencyRule`. This report indicates that there is a potential inconsistency, the name of the rule that was violated, the corresponding `ModelRelation`, and a short description explaining the issue.

Our approach is inspired by the work of Feldmann et al. [36], who define the concepts of link, linking model elements, and inconsistency, which are later utilized for inconsistency management. Our work goes beyond their approach, as we explicitly connect the concept of `ModelRelation`, which is similar to the concept of link, and `ConsistencyRule`. This emphasizes the connection between the dependent models. Furthermore, we include the management of `ConsistencyRule`s as an essential part of the consistency management, which is absent in prior work.

4. Identifying DT model consistency management requirement

Consistency management, as we have defined it so far, is a general notion for MBSs; any concrete solution that targets DTs must take the domain of DT engineering into account. In this section, we discuss the method we followed to elicit the consistency management requirements, which is depicted in Figure 4. This method is based on various requirement engineering approaches [43] and takes DT consistency management-related findings from our interview study [7] into account.

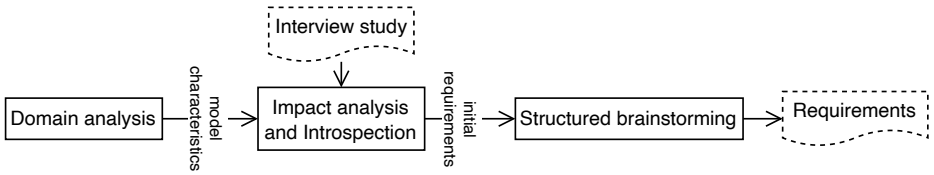


Figure 4: Elicitation method of DT model consistency management requirements.

We started with the domain analysis [44, 43] where we performed a deeper investigation into the DT models. In particular, we studied the characteristics of DT models to identify the differences and similarities with general MBSs. This is done by investigating existing literature, the process of which and the corresponding results are explained in Section 4.1.

The subsequent step involved analyzing the influence of the identified characteristics on DT model consistency management. We used a requirement elicitation technique known as *introspection* to generate a set of initial requirements. This technique is explained by Gougen and Linde [45] as the analyst relying on their own presumed knowledge of user needs. The method is generally ineffective on its own and succeeds only when the analyst is an expert in the corresponding domain and the process is facilitated by other techniques (e.g., interviews) [43]. Therefore, we integrated the key findings from the interview study with DT experts [7] and our experience as DT researchers and developers to identify the initial set of requirements, which we refined in the next phase. Here we employed another requirement elicitation technique called structured brainstorming [46], which was performed together by the authors of this paper. This method involves engaging in informal conversations and encouraging open-mindedness to quickly generate multiple ideas without focusing on any particular one. Consequently, this may lead to ideas that are unrealistic, overly ambitious, or too abstract to implement. We addressed this risk by focusing our discussion on the refinement of the initial set of requirements that had been identified in the previous step. Our expertise in DT research and engineering was essential for continuously assessing abstract ideas and ensuring that the requirements identified were practically relevant. This step led to the final set of requirements, which are explained in Section 4.2.

4.1. Characteristics of DT models

As both notions of DTs and their large-scale adoption are a recent phenomenon, there is no established and agreed upon list of such characteristics in the literature. Therefore, we analyzed over 100 publications to retrieve an overview of what the corresponding authors consider to be the characteristics of DTs. We used the keyword string *"digital twin characteri*"* (with * as a wildcard) to identify relevant literature. These were further refined by applying filters for publication date (between 2014 and 2024) and citation count (greater than 15) to identify the most recent and influential publications. We first manually skimmed the introductory parts of these papers to assess their relevancy. Thereafter, we were left with 18 papers that explicitly mentioned various characteristics of the aforementioned models or indirectly indicated them based on the corresponding use-cases. We identified eight characteristics in those 18 articles, which are shown in Table 1 and discussed here.

Characteristics	References
Frequent evolution	[47, 48, 49, 50, 51]
Heterogeneity	[11, 52, 53, 54, 48, 55]
Data-intensive	[52, 47, 56, 57, 48, 49, 1, 11, 53, 58, 59]
Physics-based	[12, 60, 55, 54, 58, 51]
Optimal accuracy or fidelity	[12, 52, 56, 49, 60, 61, 51]
Fast	[12, 53]
Trustworthy	[12, 58]
Real-time	[47]

Table 1: Characteristics of DT models identified from the literature

Frequent evolution. As the real-world counterpart evolves, the VE must co-evolve to stay aligned with it. In a DT context, the evolution of RE can be instigated, among other factors, by enhanced insights gained from simulations within VE. Moreover, meeting new requirements, bug fixes, and feature enhancements also drive the evolution of VE. Therefore, evolution is one of the key characteristics of DT models. According to Newrzella et al. [47], this evolution lasts the entire lifecycle of the DT to maintain current information at all times. Evolution is not unique to DTs as other MBSs can also evolve over time. However, this occurs more frequently for VE models within a DT. Furthermore, we also notice evolution when existing models, developed primarily for real-world entities, are reused for VE development (e.g., Walravens et al. [13]).

Heterogeneity. This is an obvious characteristic of DT models. Due to the complexity and functionalities provided, VEs can contain multiple models that encompass various aspects of the corresponding RE. For example, Tao et al. [54] mentioned four aspects: geometry, physics, behavior, and rules. These models are often cross-domain and multi-tool with divergent syntax and semantics. Therefore, heterogeneity can be recognized as a key characteristic of DT models.

Data-intensive. Data is playing an increasingly important and multifaceted role in the construction and functioning of DTs [8]. We observe data being used for optimisation, developing data-driven models, data streams being used for synchronising RE and corresponding VE, and numerous other key DT functions. Bordeleau et al. [11] emphasized the collection of data and its

use throughout the lifecycle of a DT and demonstrated the role of data with three examples. Barricelli et al. [1] discussed the usage of big data and artificial intelligence (AI) within a DT. Gargalo et al. [48] emphasized that data collection is paramount for the successful development and implementation of DTs in the bio-manufacturing industry.

Physics-based. Such models are used for twinning the physics of the corresponding RE. Wright et al. [12] discussed the importance of physics models within a VE and suggested against using non-physical models as the core of a VE. Glaessgen et al. [60] explained the use of ultra-high fidelity physics models as the backbone of the DT of air force vehicles.

Optimal accuracy or fidelity. Zhang et al. [62] listed several DT definitions ranging from cradle-to-grave model with extremely high fidelity to simplistic data-driven simulation executing in isolation. This shows the disagreement among researchers on how closely a VE should mimic the corresponding RE. However, besides differing opinions on the degree of fidelity, we observe, in most cases, a broad consensus regarding the role of fidelity within VE models [61].

Other characteristics. While only a limited group of researchers mentioned speed, trustworthiness, and real-time characteristics, we consider these highly relevant for VE models. Wright et al. [12] highlighted high-value and safety-critical systems as the most recognized areas of DT application, stressing the importance of trustworthy and fast predictions. These characteristics are also important for other use-cases (i.e., predictive maintenance). Newrzella et al. [47] emphasized the (near) real-time capability of VE models to track an RE.

Discussion. As indicated earlier, our research and the analysis of the DT model characteristics discussed here are limited to model-based DTs. As a result, the characteristics discussed are not exclusive to DTs, as other non-DT MBSs might exhibit similar properties. We further acknowledge that a concrete DT may contain models with only a subset of characteristics from Table 1. However, we argue that the co-existence of these characteristics—such as real-time bi-directional data flow, continuous synchronization with an RE, and the possibility of frequent evolution—is rare among non-DT MBSs. Furthermore, we believe that the set of characteristics as a whole differentiates DT models from the rest. Therefore, while we do not claim

completeness for the list of identified characteristics, we are convinced that they form a reasonable representation of the known characteristics of DT models. We base this on the following facts: (1) these characteristics are extracted from highly relevant, recent, and well-cited publications; and (2) the characteristics are associated with several domains and use-cases signifying their generic and inclusive nature. We also argue that these characteristics are not just descriptive but are fundamental for ensuring inter-model consistency. Therefore, to facilitate comprehensive, reusable, and unified consistency management among DT models, these characteristics must be taken into account.

4.2. Requirements for DT model consistency management

Based on the identified characteristics, we derive a set of requirements for the consistency management of DT models.

REQ01: Uniform representation of heterogeneous data. The *heterogeneous* and *data-intensive* nature of DT models points to the need for a general enough representation capable of representing syntactically diverse model information. Such a representation is essential for implementing a singular manner of analyzing the consistency of the corresponding cross-domain multi-tool models. This requires the transformation of models from their original format to the common representation. It is possible to choose one of the already existing representations as the target or develop a new one. Regardless of the choice, this adds additional overhead due to the transformation and maintenance of the common representation. However, we strongly believe that this overhead is cost-effective compared to the effort and resources needed to identify and resolve any consistency issues without an appropriate consistency management system.

REQ02: Flexible and efficient data storage. The characteristics discussed in Section 4.1 suggest that throughout the lifespan of a DT the associated *heterogeneous data-intensive* models frequently undergo *evolution*. In this context, it is crucial to choose an appropriate storage method for the uniformly represented data, as discussed in *REQ01*. Furthermore, transforming models from their original format to a unified representation has its challenges (e.g., mismatched or missing attributes, semantic issues). These factors must be taken into account when choosing the storage method. Additionally, since DTs are *real-time, data-intensive* systems, the data storage chosen for the

corresponding consistency management must be efficient in terms of space and read-write performance.

REQ03: Management of consistency rules. As explained in Section 3, consistency management is performed by defining consistency rules and checking them each time the DT or part of it evolves. These consistency rules play a central role in consistency management, forming the set of requirements that a DT must meet at all times for its proper functioning. Therefore, the envisioned consistency management must support an intuitive and efficient method for the creation, modification, and deletion of consistency rules. Moreover, the frequent evolution of models contained within a DT can render existing consistency rules irrelevant and obsolete, obstructing the overall effectiveness of the consistency management process. In essence, the proficiency of this process is fundamentally intertwined with the validity and effectiveness of the corresponding rules. Therefore, evaluating the validity and applicability of the defined consistency rules must be an integral part of consistency rule management. In addition, it is essential to reduce the overhead associated with rule management in the entire process.

These requirements are useful for general MBS, which might have similar characteristics and, therefore, similar requirements for consistency management. However, they are essential prerequisites for any system aimed at the consistency management of DT models, due to their connection to the characteristics derived above.

5. Consistency management framework for DTs

In this section, we present our consistency management framework based on the requirements derived above. In particular, this framework addresses the heterogeneity and diversity in terms of the ideation and implementation of DTs. This is achieved by defining entities that are essential for effective consistency management and their intended functionalities. The actual implementation of the framework is DT specific and may differ significantly across DTs. The framework is intended for structural consistency management and covers four core functionalities, which we describe in the next four subsections: read and extract model data; store the extracted data; evaluate consistency rules; and manage consistency rules. Figure 5 provides an overview of the framework.

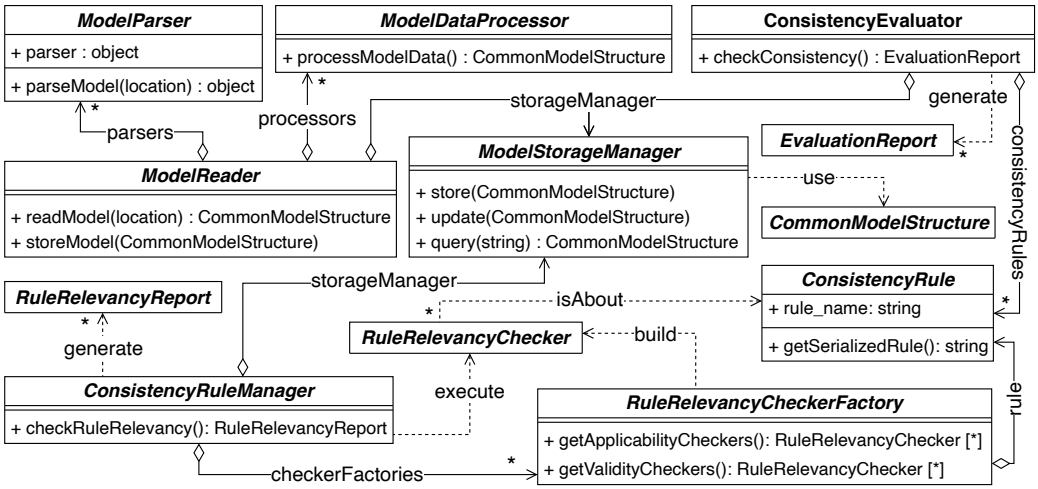


Figure 5: Consistency management framework for DT models.

5.1. Read and extract data from models

This functionality is fulfilled by the three classes described below. We define these as abstract classes that require a modeling environment-specific implementation. These environments are used to construct models contained within a DT. Since the selection of modeling environments may vary for each DT, the corresponding implementations may also differ.

`ModelParser` is responsible for parsing and extracting raw structural data from a model. As this class is the only entity directly interacting with the models, it represents the connection between the models and our framework. The parsing is done with the `parser` object, which is specific to and utilizes the interface provided by a modeling tool.

`ModelDataProcessor` processes the parsed raw data and transforms it into `CommonModelStructure`, which is the common representation (cf., *REQ01* in Section 4.2) of the heterogeneous models related to the corresponding DT. `ModelDataProcessor` is also responsible for any other post-processing of the raw model data. This can include but is not limited to removing redundant and unneeded data simplifying the `CommonModelStructure`, identifying and handling invalid data preventing further errors, and sanity checking of the resulting instance of the `CommonDataStructure`.

`ModelReader` manages the whole process of reading the models using appropriate `ModelDataProcessor` and `ModelParser` classes, and storing the resulting `CommonModelStructure` using the `ModelStorageManager`. For this

purpose, it maintains the knowledge about the models used in a DT, the modeling environments they were built in, and the parser and processor classes needed to interact with those environments.

5.2. Store extracted data

The storage of the extracted model data is managed by the `ModelStorageManager`. This class is responsible for two core functions: (1) provide an interface to the CRUD (i.e., create, read, update, and delete) services of the underlying storage solution (e.g., Neo4j[1]); (2) ensuring adherence to the `CommonModelStructure` for the stored data. As an interface to the underlying storage, `ModelStorageManager` decouples the storage from consistency management. Therefore, switching between storage solutions is possible by altering the implementation of the `ModelStorageManager` without affecting other parts of the framework.

We discussed the importance of a uniform representation of heterogeneous data (cf., *REQ01* in Section 4.2). Defining a global unified representation that supports all major modeling formalisms is a challenging task; however, it is not necessary either, as any particular DT only uses a limited set of formalisms. Therefore, our consistency management framework uses `CommonModelStructure` that unifies the formalisms used within a DT. This representation forms the schema of the stored model data and establishes the basis for defining the `ConsistencyRule`.

5.3. Define and evaluate consistency rules

The definition of consistency management presented in Section 3 includes the concept of `ConsistencyRule`, which is reused within the consistency management framework (cf. Figure 5). `ConsistencyRuleEvaluator` is responsible for evaluating all these rules and generating `EvaluationReports`, which is analogous to the `ConsistencyReport` from Section 3.

5.4. Consistency rule management

The concept of consistency rule management is introduced as one of the core activities of consistency management in Section 3. This concept is realized in the framework with the class `ConsistencyRuleManager` that executes `RuleRelevancyCheckers` to generate `RuleRelevancyReport`. These

[1]Property graph database https://neo4j.com/product/neo4j-graph-database

reports indicate whether a rule might have lost relevancy with the current version of the models (e.g., after an update) and a short description with additional information mentioning the related modeling elements for traceability. The `RuleRelevancyChecker` class has a many-to-one relationship with `ConsistencyRule`. This allows a `RuleRelevancyChecker` to analyze exactly one `ConsistencyRule`, but multiple `RuleRelevancyChecker`s can be applied to the same rule.

Each `RuleRelevancyChecker` is instantiated by a corresponding factory class [63] of type `RuleCheckerFactory` that has two separate methods for constructing two types of `RuleRelevancyCheckers`: *applicability* and *validity* `RuleRelevancyChecker`s. Each of the former is responsible for checking the *applicability* of a certain rule. In this context, we define *applicability* checking as any analysis of the rules at the meta-level. This can include, for example, the conformance checking of the `ConsistencyRule`s to the `CommonDataStructure`, and calculating the percentage of classes and properties from the `CommonModelStructure` that each rule covers. Such checkers can additionally serve to detect rules that may be in conflict with one another, for example when two or more rules specify conditions that are mutually inconsistent or produce contradictory outcomes. All other `RuleRelevancyChecker`s are classified as *validity* `RuleRelevancyChecker`. These checkers can analyze and enforce certain restrictions on a rule, control its complexity, or perform sanity checks. For example, a rule that evaluates the value of a property can have a corresponding `RuleRelevancyChecker` to verify whether the evaluated value falls within the domain of possible values for that property. This kind of `RuleRelevancyChecker`s is often implemented in the context of, and therefore strongly coupled with a `ConsistencyRule`. The *applicability* `RuleRelevancyChecker`s, however, are not closely associated with any particular rule, as they perform their analysis at a higher abstraction level. As a result, these are often reusable for analyzing multiple rules.

In our framework, we differentiate *applicability* `RuleRelevancyChecker`s from the rest, which is also apparent from the above discussion. Our motivation for this design choice is two-fold: (1) we aim to explicitly address frequent evolution of domain-specific models, their language definition, and meta-models, a phenomenon that is well known in the literature [41, 42, 4] and a key characteristic of DT models (cf. Section 4.1), which can potentially invalidate existing rules; (2) further analysis or execution of the rules that are not sound on the meta-level is pointless.

6. Implementation template

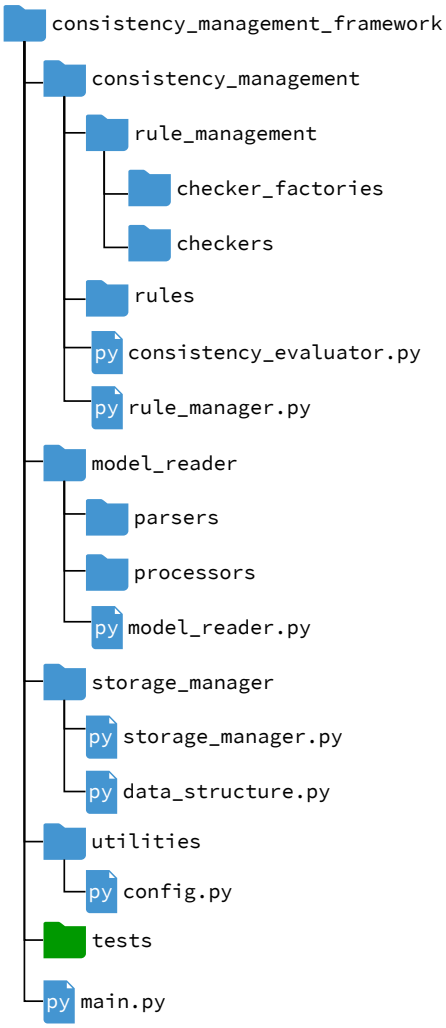


Figure 6: Python-based directory structure of the implementation template. The directories are python packages. Corresponding `__init__.py` files are not shown here.

In addition to the consistency management framework presented in the previous section, we also developed a Python-based[2] implementation template encapsulating our recommendation for structuring the implementation

[2]Python programming language https://www.python.org

artifacts (e.g., source code, configurations, and so on). This also facilitates easier and predictable navigation for the implementations that follow our recommendation. Figure 6 shows the directory structure of the template that uses Python packages and modules. However, this can be replicated with most mainstream programming languages using similar concepts.

In the directory structure depicted in Figure 6, we name the files and directories based on the names used in the consistency management framework presented in Figure 5. This allows for easier and intuitive mapping between the two. The consistency rules, their relevancy checkers and corresponding factory classes, and the evaluator classes are stored under the `consistency_management` package. The model reader, parsers, and processor classes are stored under the `model_reader` package. It is important to note that parsers are often implemented using the API provided by various modeling tools. The programming language used for this can be different from the language used for implementing the consistency management framework. Our recommendation is to store the parsers and their invocation scripts under the `model_reader/parsers` package, even if the parsers are written in a different programming language. However, if this implementation is complex and large, we recommend storing a symbolic link to it. This ensures easier navigation and keeps all the related artifacts within a single directory structure.

In our template, we store the definition of the `CommonModelStructure` in the `storage_manager/data_structure.py` module and configuration parameters are available at the `utilities/config.py` module. These parameters are static values or functions that calculate or retrieve these values from elsewhere. We strongly recommend implementing unit tests [64], which can be stored under the `tests` directory, for testing the relevant parts of the consistency management framework implementation.

7. Intended framework user and usage

The primary goal of the consistency management framework is to guide the implementation of a consistency management system that is able to address the requirements mentioned in Section 4.2. The implementation and maintenance of the framework and usage of the resulting consistency management system are a collaborative effort of the system *developers* and the *modelers*, the primary roles interacting with the system. In this section, we

briefly discuss our view of these users and the usage of the framework from a semi-technical point of view, which is also depicted in Figure 7.

As mentioned earlier, our consistency management framework, presented in Section 5, is a conceptual one. In addition, we provide an implementation template (cf. Section 6), which, in conjunction with the framework, is intended to be used as a structured approach for organizing and guiding the implementation of the consistency management system for a specific DT. Additionally, we provide three concrete working examples (see Section 8) to demonstrate the framework's implementation and usage.

Earlier in this section, we mentioned two roles, *modeler* and *developer*, that are the primary stakeholders of our consistency management framework and its implementation.

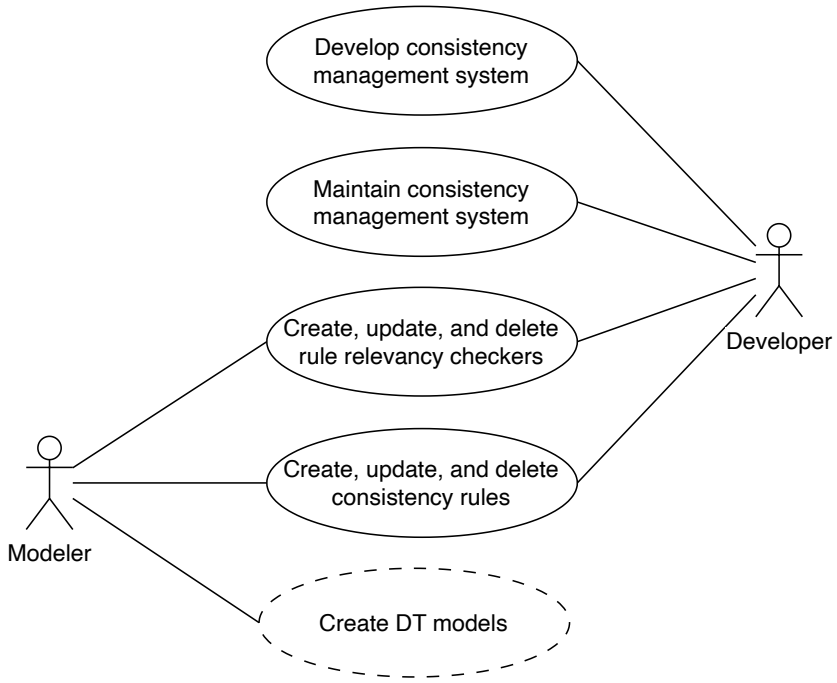


Figure 7: Framework users and their activities, represented with solid ovals, within the consistency management framework. The dashed oval shows activity outside of the framework.

Developer This role performs the tasks related to the development and maintenance of the actual consistency management system following the concepts and structures defined in our framework. In most cases,

software developers are more suited for this role as they are the ones with sufficient relevant technical skills. Additionally, they might provide technical assistance to the *modelers* in implementing consistency rules and their checkers. We foresee more involvement of the *developers* for the latter, as the checkers are not just about the models: they can be implemented to perform meta-analysis of the rules or to process the stored data. Here the developers can focus on solving challenges that are more related to software development.

Modeler This role is often filled by the domain experts who are responsible for developing or have significant influence in the development of the domain specific models, consistency management of which is the primary objective of our framework. Therefore, they possess the expertise necessary for creating effective consistency rules, although the actual implementation may be carried out by a *developer*. Similar is also applicable for the rule checkers. Furthermore, the reports produced by executing the consistency rules are also intended for the *modelers*.

8. Working examples

In the following, we present three working examples that implement our consistency management framework. With these, we aim to (1) evaluate the framework's ability to fulfill the consistency management requirements listed in Section 4.2; (2) demonstrate the framework's versatility across different domains; and (3) show that the framework can be implemented using a variety of technologies. Therefore, the working examples cover multiple domains—autonomous vehicles, language workbenches, robotics. Furthermore, we implemented them using different general-purpose technologies that are relatively well-known and used across various domains. This is to demonstrate that the framework requires no specialized or theoretical background to implement. This can facilitate broader adoption and allows developers and organizations to leverage their existing skill sets and infrastructure, reducing the time and resources needed for implementation.

The working examples are based on two different graph-based storage technologies as their foundation—Neo4j and Resource Description Framework (RDF)[3]. We use graph-based storage technologies because of their flex-

[3]Resource Description Framework https://www.w3.org/RDF/

ibility and capability to efficiently store and query highly interconnected data. The following discussion is divided on the basis of the storage technology used in the working examples.

8.1. Working examples using Neo4j

In this section, we present two working examples with Neo4j-based implementations of our consistency management framework introduced in Section 5. The first is to analyze the consistency among the models of a DT of an autonomous truck capable of docking itself at a distribution center [65]. The second analyzes the consistency of textual grammar-based DSL definitions (i.e., Xtext[4]). For this purpose, we use the Xtext-based pilot implementation of the second version of the Systems Modeling Language (SysML-V2) [66].

The implementations of both working examples share numerous similarities in terms of activities and technologies used. The majority of the implementation of the working examples is done in the Python programming language using the implementation template presented in Section 6. For data storage, we use Neo4j, a labeled property graph database, due to its capability to efficiently manage data that lacks strict structuring and is highly interconnected. Its query language, Cypher [67], is highly expressive and intuitive, allowing a visual overview of the queried pattern within the stored graph data. We use this to write `ConsistencyRule`s, which are essentially inconsistency patterns expressed in Cypher.

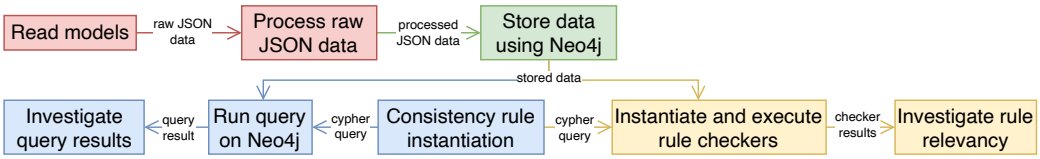


Figure 8: Overview of consistency management activities performed in the working examples presented in Section 8.1. The relation between these activities and key framework functionalities are indicated using colored boxes. Activities highlighted in red, green, blue, and yellow correspond to the four core activities of our framework explained in Sections 5.1, 5.2, 5.3 and 5.4, respectively. Due to data and process dependencies, the sequence must be red first, then green, followed by blue and yellow, which can occur in any order or in parallel after green.

Figure 8 shows an overview of the activities performed within working examples and their relationship to the key functionalities of our consistency

[4]Language workbench Xtext https://eclipse.dev/Xtext/

management framework (cf. Section 5). We start by extracting information from the models and storing them as JSON files. These files allow cross programming language interfacing as the programming interface language varies across modeling tools. Afterwards, we further process this raw data to (1) convert them to a `CommonModelStructure`, which is explained in Section 5.1; (2) establish connections between different model entities, especially in cross-tool scenarios, where relationships may not be explicit; and (3) investigate and fix any faulty data. In the following step, we store the processed data using Neo4j. We use the nodes and edges in the graph data to respectively represent various modeling entities and relations among them. This structure is the manifestation of the concept of `CommonModelStructure` and varies between working examples (i.e., as in Figures 10 and 15). Once stored, this graph data can be queried for patterns that are useful to ensure consistency between the corresponding models. In the working examples the `ConsistencyRule`s are essentially Cypher queries each representing a specific possible consistency issue. In the next sections, we present examples of these generated queries for the corresponding working examples. Finally, we investigate the query results and related model entities to identify any consistency issues.

The consistency rule management (cf. Section 5.4) related activities are highlighted yellow in Figure 8. `RuleRelevancyChecker`s are instantiated and executed to check the validity of various consistency rules, generating relevant reports which are manually investigated to identify any issues within the rules. The instantiations take place within various factory classes, each of which is associated with a consistency rule. These factory classes possess the knowledge of which checkers are needed to analyze their respective rules. Unlike `ConsistencyRule`s, the `RuleRelevancyChecker`s are not just encapsulated Cypher queries. They are often a mixture of complex logic written in Python checking various conditions a rule must fulfill and, if needed, one or more Cypher queries that extract necessary information from the stored data. To perform rule-checking at the meta-level (i.e., applicability `RuleRelevancyChecker`s), we use ANTLR4 [68] to construct and analyze the abstract syntax trees (AST) for the associated Cypher queries. In the following sections, we present concrete examples of applicability and validity `RuleRelevancyChecker`s implemented within the working examples.

8.1.1. Consistency management of truck docking DT

In this working example we manage the consistency of the models contained in the DT of an autonomous truck, which is capable of autonomously driving and docking itself within a virtual distribution center [65]. The VE is created by integrating Simulink (a simulation environment), IBM Rhapsody (a SysML modeling tool), and Unity (a 3D game engine) models. The behavior and structure of the distribution center and the autonomous truck are modeled in Rhapsody. The Simulink model handles the truck's path planning, kinematics, and collision detection. The Simulink models are imported into Rhapsody as `Blocks`, with their `Ports` mapped to corresponding `Ports` in the Simulink model. The 3D environment including the physics and interactivity is implemented in a Unity model. This working example focuses on the consistency management between the Simulink and Rhapsody models. Figure 9 shows an overview of the implementation of our consistency management framework for this working example.

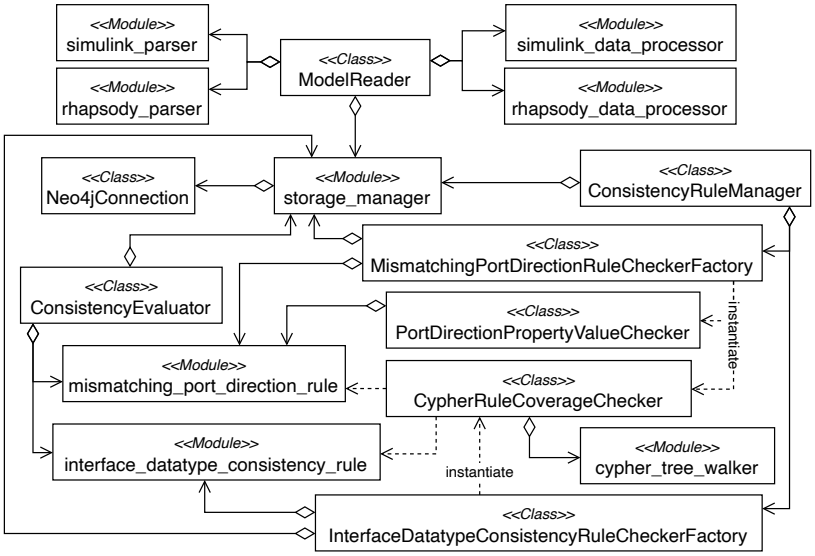


Figure 9: Implemented entities of the consistency management framework for autonomous truck DT [65]

Extract and store data from models. This is orchestrated by the `ModelReader` class that uses two sets of parsers and data processors to extract structural information from the Simulink and Rhapsody models. The data processors ensure the conformance of the processed data to the meta-model presented

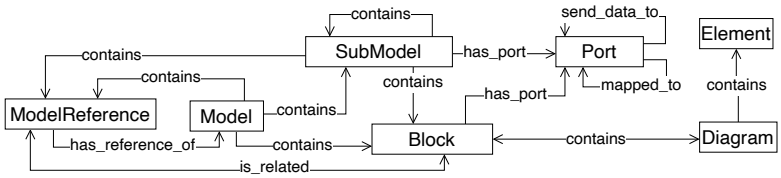


Figure 10: Excerpt of the `CommonModelStructure` for Simulink and Rhapsody models in truck docking DT

```
MATCH (smln_mdl:Model)-[c: CONTAINS ]->(sb:Block)
    -[hp:HAS_PORT]->(smln_prt:Port)
MATCH (rpsd_prt:Port)-[m:MAPPED_TO]->(smln_prt)
WHERE rpsd_prt.data_type is null
    OR smln_prt.data_type is null
    OR rpsd_prt.data_type <> smln_prt.data_type
RETURN rpsd_prt, smln_prt, sb, smln_mdl
```

(a) Cypher query

```
Simulilnk model: PathPlanning
Simulink block: Kp_dpsi
Simulink port path: PathPlanning/Kp_dpsi/Inport1,
    data_type: unspecified
Rhapsody port path: Default::PathPlanningSim.Kp_dpsi,
    data_type: unspecified
```

(b) Excerpt of query result in plain text

Figure 11: The Cypher query encapsulated by `interface_datatype_consistency_rule` is shown in (a) that check the inconsistency or unavailability of data type between mapped Simulink and Rhapsody ports. An excerpt of the query results, formatted as plain text, from executing (a) is presented in (b). This excerpt highlights a port pair where the data type is unspecified.

in Figure 10, which is the `CommonModelStructure` for this working example and is created by merging similar concepts from both modeling tools. For example, in both modeling environments, a model can refer to external models, and we encapsulate this concept as `ModelReference`. Here, the concept of similarity is determined by the modeler implementing the framework, depending on the context.

Consistency evaluation. The `ConsistencyEvaluator` class is responsible for evaluating all `ConsistencyRules`. At the bottom left corner of Figure 9 we present two such rules aggregated by the evaluator class. The rules are implemented as Python modules that encapsulate Cypher queries to identify inconsistency patterns. An example of such a query associated with the `interface_datatype_consistency_rule` is presented in Figure 11a. It checks whether the data type of a `Port` in a Rhapsody model matches the corresponding `Port` in Simulink. Running it on the data stored in Neo4j retrieves pairs of mapped ports with mismatching or unspecified data types, which might cause runtime issues. Figure 11b shows an excerpt of the query result.

```
MATCH (m1:Model)-[c: CONTAINS ]->(b1:Block)-[h:HAS_PORT]->
        (outp:Port)-[s:SEND_DATA_TO]->(inp:Port)
WHERE inp.direction = 'Outport' OR outp.direction = 'Inport'
RETURN inp, outp, m1, b1
```

(a) Cypher query for `mismatching_port_direction_rule`

```
===>Analysing rule:
    'mismatching port direction between property and edge direction'
Directions addressed in query but not in db {'Outport', 'Inport'}
Directions present in db not addressed in query {'In', 'Out'}
```

(b) Result from rule relevancy checker `PortDirectionPropertyValueChecker`

Figure 12: Figure (a) shows the Cypher query encapsulated by `mismatching_port_direction_rule`, which checks for mismatches between a port's direction property and its actual usage. Figure (b) presents the result from the rule relevancy checker `PortDirectionPropertyValueChecker`, which compares the values of the port's direction property checked by the rules and values stored in the database. In this example, some mismatch is found.

Consistency rule management. Here we present two examples of consistency rule management for this working example. Figure 12 depicts the Cypher query associated with the rule `mismatching_port_direction_rule` and the output of `PortDirectionPropertyValueChecker`. The rule checks whether the `direction` property of a `Port` is consistent with its actual usage (e.g., an `Outport` should not receive data). The checker ensures that all values of the `direction` property stored in the database are checked by the consistency rule. Figure 12b shows the result of the check which identifies several mismatches between the checked and stored values. Another example of a rule relevancy checker is presented in Figure 13, which shows the result of executing `CypherRuleCoverageChecker`. This applicability checker ensures the conformance of consistency rules to the `CommonModelStructure` (see Figure 10) by creating a mapping between the classes and properties referenced in the Cypher query of the consistency rules and those defined in the `CommonModelStructure`. An example of such mapping for `mismatching_port_direction_rule` is shown in Figure 13a. Moreover, such mappings at a meta-level can provide a high-level overview of the rule's coverage by showing the classes and their properties that are addressed in the rule. This can help modelers assess whether existing rules adequately cover the data stored. For example, Figure 13b shows the combined coverage of the rules presented in Figures 11a and 12a.

```
Block referred
Block.outputPortCount NOT referred
Block.inputPortCount NOT referred
Block.path NOT referred
Block.id NOT referred
Block.SolverType NOT referred
Port referred
Port.data_type NOT referred
Port.id NOT referred
Port.PortNumber NOT referred
Port.direction referred
Model referred
Model.path NOT referred
Model.id NOT referred
```

(a) Correlation for rule in Figure 12a

```
Block referred
Block.outputPortCount NOT referred
Block.inputPortCount NOT referred
Block.path NOT referred
Block.id NOT referred
Block.SolverType NOT referred
Port referred
Port.data_type referred
Port.id NOT referred
Port.PortNumber NOT referred
Port.direction referred
Model referred
Model.path NOT referred
Model.id NOT referred
```

(b) Combined correlation for rules in Figures 11a and 12a

Figure 13: Correlation between meta-entities addressed in consistency rules and those in `CommonModelStructure` (see Figure 10). Classes and properties appear in black, with green or red text indicating their occurrence in the query. Figure (a) illustrates the correlation for one query, and Figure (b) for two queries.

8.1.2. Consistency management of Xtext grammar definition

With this working example we perform a change impact analysis on the SysML-V2 Xtext grammar [66] supporting the maintenance of its consistency. Xtext is an open source framework for creating textual syntax for DSLs. Here we use various Xtext specific terms, explained in [69], omitting explanations for brevity. Although seemingly unrelated, we believe that this working example is relevant for DTs since (1) the usage of DSLs is increasing within the scope of DTs [9, 10, 7]; and (2) while general-purpose modeling languages (e.g., UML and Simulink) evolve infrequently, in-house DSLs in industrial settings are more frequently adapted [41, 70], increasing the likelihood of introducing inconsistencies.

Several aspects of the implementation of this working example are similar to the truck docking DT working example (cf. Section 8.1.1). The similarities in terms of the activities performed within these working examples are described at the beginning of Section 8.1. Additionally, between the implementations of the consistency management framework, similar names are used for various classes and methods in both working examples. Figure 14 shows the implementation of the consistency management framework for this working example.

Extract and store data. This process is orchestrated by `ModelReader` using two parsers (i.e., `grammar_parser` and `metamodel_parser`) and a processor (i.e., `dependency_graph_processor`). The parsers extract various enti-

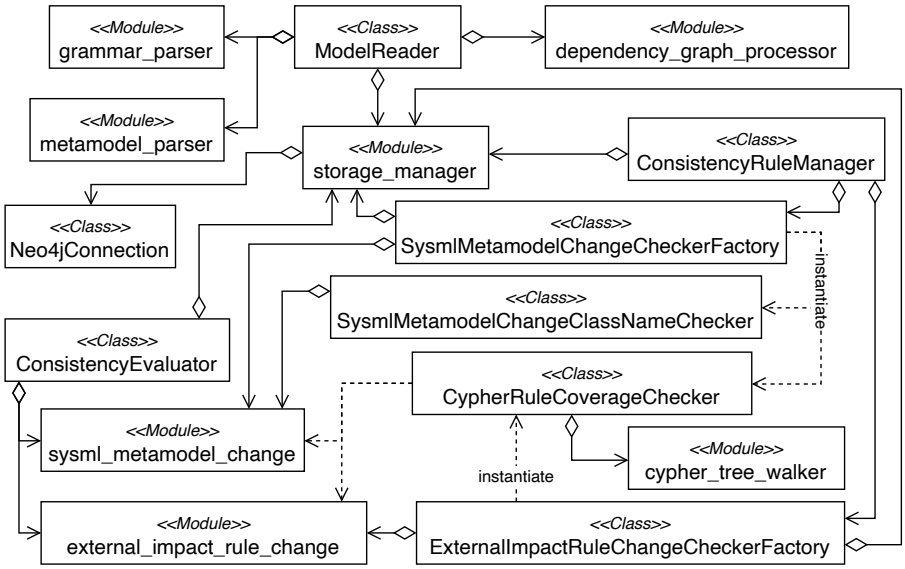


Figure 14: Implemented framework entities for the working example on SysML-V2 Xtext grammar

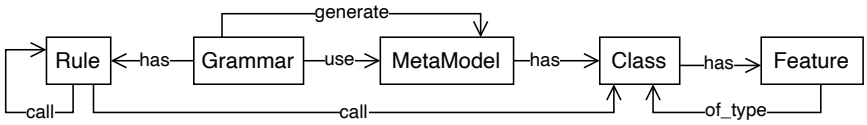


Figure 15: Excerpt of the `CommonModelStructure` for Xtext grammars and meta-models in SysML-V2 repository

ties (i.e., classes, features, and rules) and relations among them from the grammar and meta-model definitions available in the repository. These are further processed by the processor to reformat them to (1) align with the `CommonModelStructure` used in this working example; and (2) adapt it to be suitable for storage as a dependency graph.

Consistency evaluation. Similarly to the truck DT working example (cf. Section 8.1.1), the consistency rules are essentially Cypher queries that search for certain patterns within the data stored in the graph database. For example, Figure 16 shows the Cypher query for the consistency rule `sysml_metamodel_change_rule` and an excerpt of the result from executing it. This rule identifies the impacted grammar rules when modifying the `Documentation` class defined in the *sysml.ecore* meta-model. This query is the result of instanti-

```
MATCH (gm:Grammar)-[u:use]->(mm:MetaModel{
  uri: 'https://www.omg.org/spec/SysML/20240201'})
MATCH (gm)-[h1:has]->(r:Rule)-[c:calls]->(cls:Class {
  name:'Documentation'})<-[h2:has]-(mm)
RETURN r, gm, cls, mm
```

(a) Cypher query

```
Changed class: Documentation
Contained metamodel: org.omg.sysml/model/SysML.ecore
Used in grammar: org.omg.sysml.xtext/target/classes/org/omg/sysml/xtext/SysML.xtext
Rule: Documentation

Changed class: Documentation
Contained metamodel: org.omg.sysml/model/SysML.ecore
Used in grammar: org.omg.kerml.xtext/target/classes/org/omg/kerml/xtext/KerML.xtext
Rule: Documentation
```

(b) Excerpt of the result from executing the Cypher query in (a)

Figure 16: Cypher query for `sysml_metamodel_change_rule` consistency rule and an excerpt of the result from executing it. The result in Figure (b) shows two grammar rules from different grammar files that can be impacted as a result of modifying `Documentation` class in the *sysml.ecore* meta-model.

ating a template with class name `Documentation`. We can apply the same process for any other class defined in the same meta-model. This simple yet useful consistency rule can provide insight into possible impacts on grammar rules when modifying *sysml.ecore*, allowing modelers to make informed decisions and potentially avoid consistency issues.

Consistency rule management. We present two examples of consistency rule management in this working example, which are depicted in Figure 17. Figure 17a shows the results of the applicability checker `CypherRuleCoverageChecker` that validates the conformance of the consistency rule with the `CommonModelStructure` presented in Figure 15. It is done by identifying the classes and properties of the `CommonModelStructure` that are addressed in the query in Figure 16a. Figure 17b shows the result of the validity checker `SysmlMetamodelChangeClassNameChecker`. This checks whether the class name, `Documentation` in this case, actually exists within the specified meta-model. This helps a modeler locate and modify the related rule if the class is renamed or removed.

Visual analytics. In addition to the change impact analysis, this working example provided us with additional insights that can indirectly contribute to improving consistency management. Figure 18 presents a snapshot of the stored graph data, in which grammar rules, meta-model classes, and

```
MetaModel referred
MetaModel.name NOT referred
MetaModel.uri referred
MetaModel.file NOT referred
Class referred
Class.name referred
Feature NOT referred
Feature.name NOT referred
Grammar referred
Grammar.namespace NOT referred
Grammar.file NOT referred
Rule referred
Rule.name NOT referred
```

(a) Correlation between the query and `CommonModelStructure`

```
Class 'Documentation'
Contained in metamodel 'org.omg.sysml/model/SysML.ecore'
URI 'https://www.omg.org/spec/SysML/20240201'.
    Total count 1
```

(b) Result from validity checking

Figure 17: Results of consistency rule management for the rule in Figure 16a. Figure (a) illustrates the correlation between the meta-entities address in the query and those specified in the `CommonModelStructure` (cf. Figure 15). Meta-entities, like classes and properties, are shown in black, with green or red text indicating their presence in the query. Figure (b) shows the result of validity checking of the Cypher query, where the availability of the class name within the corresponding meta-model is checked.

features are represented by nodes colored yellow, red, and green, respectively. This clearly shows different clusters where nodes within a cluster are closely related. Since these clusters are independent, any changes to the entities are confined to the respective cluster, thus localizing their effects. We also notice that the *owl* grammar generates the *owl* meta-model (circled blue in Figure 18), which is the only one generated, indicating a strong probability of simultaneous changes of these entities.

8.2. Service-Oriented Robotic Cell with RDF

Our third working example is on a robotic cell with two robotic arms, and implements the DT architecture proposed by Gil et al. [71], which describes the control of the two arms using coupled DTs. The architecture contains a part for consistency management, named *defect analysis*, which we adapted to the consistency management framework proposed in this paper. Compared to the systems described in Section 8.1, this system has different settings for both the architecture and underlying graph technology: (1) it follows a service-oriented DT [72, 54] architecture; and (2) it uses the RDF technology stack as the underlying graph technology.

The robotic cell consists of two robotic arms as physical entities. There are two Functional Mock-Up Units (FMU) [73] associated with each robotic arm. One of the FMUs virtually represents the corresponding arm by simulating its behavior, and the other manages the connection to the physical en-

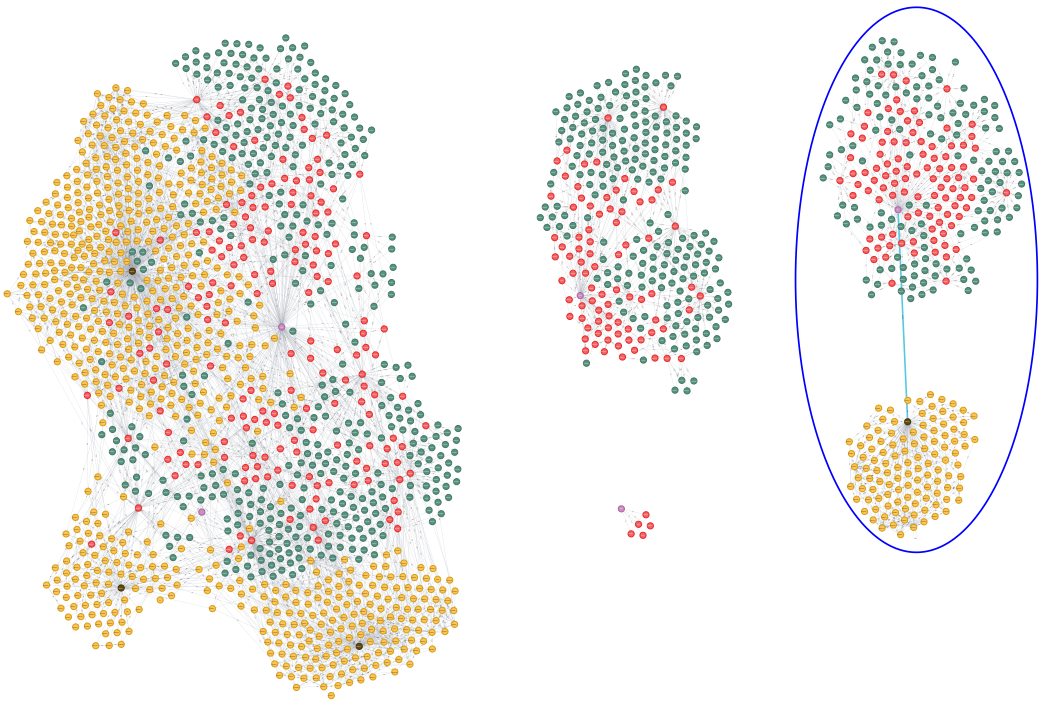


Figure 18: Visualization of SysML-V2 Xtext repository [66] data stored in Neo4j. Nodes colored yellow, red, and green signify grammar rules, meta-model classes, and features, respectively. The blue circled part shows the only case where the meta-model is generated from the grammar.

tity. Each FMU is considered a singular endpoint. The architecture provides modules to configure these endpoints, couple them using co-simulation [74], and provides a number of services that operate on the coupled endpoints. Of particular interest here is `LiftingService`, which exports the current configuration of the program [75, 76] as a knowledge graph. The configuration contains all components of the software, as well as their current states. For example, it contains the FMUs [16], as well as all objects that instantiate the classes of the software architecture. We refer to the knowledge graph exported from a configuration as the *lifted configuration*.

Extract and store data from models. The instantiation of our framework for this system is shown in Figure 19. The `CommonModelStructure` (CMS) is implemented as an `FMUEndpoint`. Each `FMUEndpoint` is a wrapper for the used FMUs, together with additional information, such as the name of the RE, a CMS for a VE it is twinning (using the `twinnedWithName`

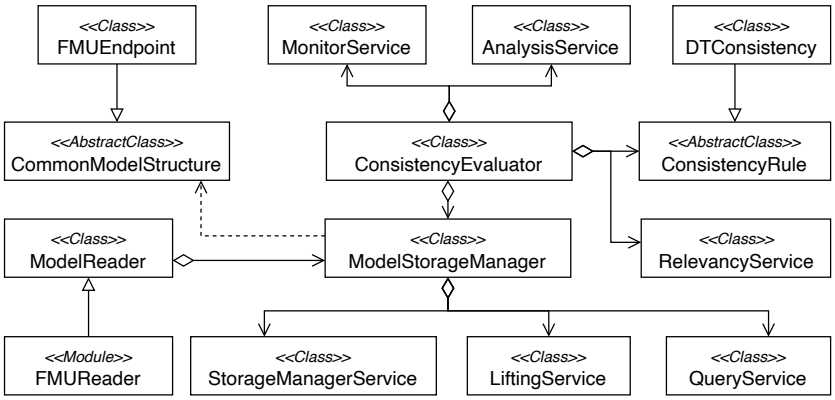


Figure 19: Instantiation of the consistency management framework for the robotic cell.

field). But the `ModelParser`, `ModelReader`, and `ModelDataProcessor` are external to the system – as the considered models are all FMUs, they are handled by the used FMU library, which can parse FMU files, processes the raw data and makes available by providing an API that realizes co-simulation, thus playing the role of `ModelParser`, `ModelDataProcessor` and `ModelReader`. The `ModelStorageManager` is a composite of three services: the `StorageManagerService` that stores and updates CMS, and the aforementioned `LiftingService` and `QueryService` that realize the `query` function. The last two services can also be used for other purposes than consistency checking (e.g., lifting is a form of serialization and querying the lifted configuration can be used for self-adaptation [75]).

Consistency evaluation. Consistency is defined using *defect queries*, which are SPARQL[5] queries to identify inconsistency patterns. A configuration is considered consistent if all defect queries on the lifted configuration return an empty result set, i.e., if there is no instance of an inconsistency pattern. A `ConsistencyRule` is a pair of a defect query and a call-back function.

The `ConsistencyEvaluator` is again a composed service, consisting of the aforementioned `MonitorService` and `AnalysisService`, as well as the `RelevancyService` which we describe further below. This is realized by the `QueryService`, that allows to run SPARQL queries on the lifted configuration, the `MonitorService` that repeatedly evaluates a defect query and

[5]SPARQL query language https://www.w3.org/TR/sparql11-query/

```
SELECT ?id ?idNext {
  ?x a SimulationComponent; hasName ?id.
  ?asset twinnedWithName ?id.
  ?assetN twinnedWithName ?idNext.
  ?cont1 a ContainerComponent;
         contains ?asset; contains {twinnedWithName ?idNext}
         hasConnection [
           connectFrom [partOf ?asset];
           connectTo [partOF ?assetN]].
  FILTER NOT EXISTS {
  ?y a SimulationComponent; hasName ?id.
  ?cont a ContainerComponent;
        contains ?x; contains ?y;
        hasConnection [
          connectFrom [partOf ?x];
          connectTo [partOf ?y]] } }
```

Figure 20: Defect query for checking consistency among four FMUs.

issues a callback to the `AnalysisService` for each found defect. The analysis service then generates a report based on the retrieved defect.

Figure 20 shows an example of a defect query that checks the consistency between four FMUs: two FMUs model the REs, while two model the VEs. Inconsistency would occur if the `Connection` between the REs would not be mirrored by a `Connection` between the VEs. The first lines retrieve the first VE-FMU (`?x`), the RE-FMU that corresponds to it (`?asset`) and the next RE-FMU (`?assetN`). In case a defect is detected, the call-back function of the rule is invoked, and a consistency report is generated that contains the result of the query and some information. Figure 21 shows an example of such a report (where `?id` is `sim_1` and `?idNext` is `sim_2`).

```
The following simulators not connected correctly: sim_1, sim_2
The PE of sim_1 is connected to the PE of sim_2,
      but sim_1 is not connected to sim_2
```

Figure 21: Example of a report created based on the results of executing the *defect query* in Figure 20.

Consistency rule management. Relevancy is managed by the `RelevancyService`. The relevancy service is invoked by the `ConsistencyEvaluator` upon changes to the `ConsistencyRule`s, or their addition. For each `ConsistencyRule`, its relevancy is determined by retrieving the *non-optional* sym-

bols in the contained query, such as concept and relation names, and checking that they are all used in the lifting configuration. For example, the retrieved symbols in our example query are `SimulationComponent`, `hasName`, `twinnedWithName`, etc. Some symbols are excluded because they are independent of the DT, such as the relation expression set membership "`a`". If a symbol is not used in the lifted configuration, then the query will always return an empty set, not because there is no defect, but, e.g., because the ontology used for lifting was changed and drifted from the ontology used in the defect query; drift is a change in the ontology or meta-model, while a defect is in a graph realizing the ontology.

There were no major difficulties in adapting the architecture of Gil et al. to our consistency management framework, as all services and data structures (such as FMUs) mapped well to it. The biggest change is to introduce an explicit notion of `ConsistencyReport` and to introduce the components necessary for relevancy, which the previous architecture did not consider.

9. Consistency management framework evaluation

In Section 4.2, we defined three requirements for any system aimed at consistency management of DT models. In the following, we evaluate our consistency management framework and its implementation in the working examples from the perspective of these requirements.

REQ01: Uniform representation of heterogeneous data. To address the need for a general representation of heterogeneous models and their data, our consistency management framework includes the concept of `CommonModelStructure` (cf. Section 5), which is a uniform representation of data extracted from heterogeneous models used within a particular MBS.

We also make the concepts of parser and processor explicit in the framework. These can have custom implementations targeting specific modeling formalisms that allow for the extraction and processing of structural data from the corresponding models. Although it may take considerable time and effort to implement these, we argue that the investment is typically a singular occurrence since the modeling formalisms utilized within the project context rarely undergo frequent changes. This method makes the framework adaptable and applicable across a wide variety of projects. Our working examples (cf. Section 8) are perfect examples of the implementation of concepts related to data extraction and storage, as these working examples are highly diverse

in terms of use-cases and used modeling technologies. These also demonstrate the feasibility of developing a unified representation for models (i.e., `CommonModelStructure`) created from more than one modeling formalism.

REQ02: Flexible and efficient data storage. In our case studies, we use graph-based storage (i.e., Neo4j and Apache Jena/RDF), which is an established technique for large-scale processing of complex and interconnected data [77, 78, 79, 80, 81, 82]. Based on these works and our working examples, we believe that graph-based data representation and storage have great potential for the consistency management of DT models and, therefore, addressing *REQ02*. This potential has also been recognized in various existing research works, as presented in Section 2. Furthermore, our framework includes the concept of `ModelStorageManager` that abstracts the low-level storage and provides a standard interface to interact with it. This allows for further flexibility by decoupling the consistency management system from the lower-level storage solution.

REQ03: Management of consistency rules. This requirement establishes the need for the management of consistency rules, which includes two main activities: (1) create, update, and delete consistency rules; and (2) evaluate the validity and applicability of the defined consistency rules. These are fully supported within our consistency management framework using concepts such as `ConsistencyRule` and `RuleRelevancyChecker`. The former plays a central and multifaceted role in our consistency management framework by explicitly encapsulating the nature of relationships among models within an MBS and preserving them in an environment where models can undergo frequent changes. Our working examples show several examples of such rules and results from executing them, demonstrating their effectiveness. We also present example implementations of `RuleRelevancyChecker` that analyze the corresponding rules and provide additional information on their validity and applicability.

The analysis in this section shows that our consistency management framework, introduced in Section 5, addresses the requirements identified in Section 4.2, thus being capable of managing consistency rules to maintain consistency among DT models. Furthermore, based on this discussion and the working examples presented in Section 8, we conclude that this framework is flexible enough to be implemented in a wide range of projects employing various modeling environments, potentially extending even to settings that

are not DT-based, and that the use of graph-based storage techniques is suitable in this context.

10. Practical usage: insights from the working examples

This section outlines the practical challenges and considerations for implementing and maintaining a consistency management system that implements our framework. These are derived from the insights gained from the working examples discussed earlier.

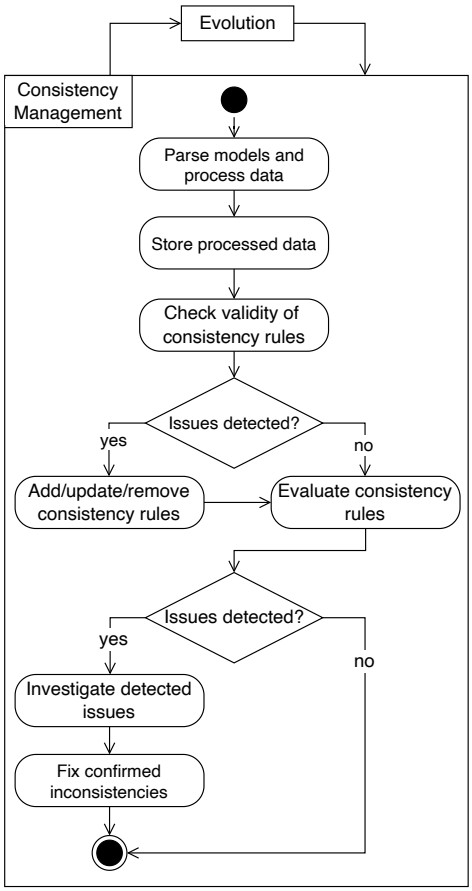


Figure 22: Recommended DT model evolution and consistency management cycle with detailed consistency management activities.

Application in DT lifecycle In Section 4.1 we discussed that evolution is a key characteristic of DTs and the models contained within. We

recommend that the consistency management systems be primarily applied at every stage of evolution, as depicted in Figure 22. It shows that every evolutionary step is followed by the consistency management activities, which are illustrated in detail. The goal is the fast and effective identification of any new inconsistencies that may have arisen within the DT as a result of changes in the models, while maintaining the relevance of the consistency rules with the corresponding models at all times. In a version controlled (i.e., Git[6]) continuous integration and deployment (CI/CD) setting [83], the consistency management activities can be part of the CI/CD pipeline.

Usage of rule evaluation and relevancy report Depending on how the rules and rule checkers are implemented, the resulting reports can identify issues that may pose potential problems or definite issues requiring immediate attention. In the case of potential problems, a human must be in the loop to investigate, confirm, and address the issue as needed. Our experience from the working examples suggests that defining rules and checkers to definitively detect an issue is challenging, as its future manifestation is often difficult to predict. The working examples in Section 8 present several example reports from rule evaluation and validity checking, most of which indicate potential issues. While these reports may not definitively signal an error, they significantly reduce the search space for finding one. In a large enough project, this capability can dramatically decrease the effort and resources needed to maintain model consistency.

System maintenance This is about maintaining the implemented consistency management system by modifying or extending the existing implementation. This activity can be triggered by an update to the corresponding DT, such as the addition of new models that require new parsers and processors (cf. Section 5.1). Similarly, it can occur when models previously excluded from the consistency management system need to be incorporated. Furthermore, like any software-based system, a consistency management system might require bug fixes, performance improvements, dependency upgrades, extensions, and so on, all of which are necessary for maintaining an effective system.

[6]Git version control system https://git-scm.com/

Consistency rule maintenance This is directly related to the consistency rule management explained in Sections 5.3 and 5.4. The `ConsistencyRules` need to be created according to the consistency management specifications of the DT. These rules must be modified whenever necessary and deleted if they become obsolete. The framework offers some support in managing the rules through the implementation of `RuleRelevancyCheckers`, which can detect irrelevant rules (cf. Section 5.4), as demonstrated with our working examples (cf. Section 8). However, verifying the results of the checkers and the actual update, as indicated earlier, requires manual effort.

11. Threats to validity

In addition to the evaluation presented in Section 9, we critically analyzed our consistency management framework and identified three potential threats that challenge the validity of our approach. In this section, we explore and discuss these scenarios. The improvement possibilities of the framework and further research directions related to these threats are discussed in Section 12.

TV01: Small-scale working examples and framework evaluation. With the three working examples (cf. Section 8) presented in this paper, we demonstrate the functionalities of our consistency management framework and argue its broad applicability (cf. Section 9). However, these working examples are relatively small in nature, limiting the evaluation of the framework in terms of the practical applicability, kinds of inconsistencies it is able to handle, modeling environments that can be supported, data limitations, efficiency, etc. This is a threat to the validity of our conclusions.

TV02: Noncentralized knowledge of model structure. The unified representation of heterogeneous model data is a key functionality of our consistency management framework (cf. Section 5.1). To achieve this, any data extracted from the models undergoes multiple processing steps, which are demonstrated in the working examples. These steps can include reformatting and converting the original data to a desired representation before storing it. The consistency rules and rule relevancy checkers, which form the backbone of our consistency management framework, are created based on the `CommonModelStructure`, the processed model data introduced in Section 5.1. Therefore, creating meaningful and context-specific consistency rules and rule relevancy checkers requires a detailed understanding of the

relationship between the transformed and original model data. This can complicate the consistency management process, particularly in large, multi-domain projects where all the necessary knowledge is often distributed across various domain experts. As a result, defining consistency rules and rule relevancy checkers remains a challenging task, especially for larger projects, threatening the validity of our approach.

TV03: Challenges in unifying model representations. We recognize that developing a unified representation for models remains challenging, as demonstrated by the significant effort required to standardize the FMI interface for simulation models, particularly when the modeling formalisms differ substantially from each other. Although this has the potential to block further implementation of the framework, it is a general problem in the field of model-based methods (and the subject of many industrial and research activities). However, in the scope of single or internal projects, our framework remains applicable, and in the case of larger projects, we conjecture that conceptually dividing the complete system into smaller subsystems and implementing consistency management for these subsystems may result in better scalability.

TV04: Inappropriate method for consistency management requirement elicitation. In Section 4, we described the method for eliciting the consistency management requirements for DT models. Since the authors of the paper are also the ones acting as the analyzers and elicitors, the results are at risk of subjectivity. In essence, our experience and expertise might have unintentionally led us to favor or interpret the requirements such that they align with our pre-existing beliefs about DTs, instead of reaching objective conclusions. To minimize this risk, used information from external sources such as the interview study [7] and the literature studied in Section 4.1. We also employed a multi-step elicitation method that allowed us to gather and utilize information from these external sources and refine the requirements in multiple steps.

12. Future research

The threat to validity *TV01*, discussed in the previous section, can be addressed by performing further working examples implementing our consistency management framework for larger DT projects across domains not addressed in this article. Such working examples can provide scientific value

in two ways: (1) further validation of the applicability of the framework for additional use-cases and domains; and (2) identify potential improvement possibilities to ensure even broader applicability in cases where it is not implementable. Although the working examples provides us with insights into the frameworks' applicability, feasibility, and capabilities in managing model consistency (cf. Sections 9 and 10), our current research does not assess the framework's performance or efficiency. This is because it is a conceptual framework, and evaluations of this kind are only possible on tangible implementations. We understand that the design choices of the framework might have some effects on the performance of the implementations, identifying which remains open for further research.

`ConsistencyRule`s play a central role in the detection of inconsistencies. Maintaining a set of well-defined and relevant rules is essential for ensuring the effectiveness of the consistency management system. Therefore, we included the management of `ConsistencyRule`s as an integral component of our framework and demonstrated examples of it in the working examples. However, due to the conceptual nature of the framework and the relatively limited scope of the working examples, multiple aspects of `ConsistencyRule` management and its practical implementation are not yet fully defined, presenting an opportunity for further exploration. These aspects are related to the ways to define the `ConsistencyRule`s, the cost associated with their maintenance, and the overall gain in terms of effort and time. Threat *TV02* calls for the development of an appropriate method for defining consistency rules and rule relevancy checkers. Several methods for defining consistency rules have been proposed in the current literature. For example, Feldmann et al. [31] proposed using triple graph grammars for this purpose. In our working examples, we used Cypher and SPARQL queries to achieve the same goal. However, none of these approaches can sufficiently address the challenge explained in *TV02*, pointing to the need for further research. This is especially significant for DTs, considering their extensive acceptance in recent years and the catastrophic consequences that inconsistencies can have on them. The scalability and run-time performance of executing the rules and the corresponding checkers on larger-sized projects also remain to be evaluated. Furthermore, future research can focus on understanding what specific information needs to be included in the reports generated by the rules and checkers to effectively help developers identify and fix issues in the models and the rules. This includes determining the content that is both useful and sufficient to enhance and fast-track the process.

We define the concept of `RuleRelevancyChecker` in our framework as a mechanism to manage the `ConsistencyRule`s (cf. Section 5) and demonstrate their exemplary use throughout our working examples. These examples, however, should be viewed primarily as indicative applications rather than as an exhaustive account of the checkers' capabilities. We believe that when carefully designed and properly implemented, such checkers have the potential to support a far broader and more sophisticated analysis of the rules than what our current working examples are able to show, including but not limited to conflicting rule identification and suggesting adaptations of `ConsistencyRule`s based on evolving `ModelRelation`s. Therefore, we suggest further investigation into the capabilities and improvements of the concept of `RuleRelevancyChecker`.

In threat *TV03* we indicated that creating a unified representation for several modeling formalisms can be challenging and resource-intensive. We believe that developing a (semi-)automated method for generating a mapping from a specific modeling formalism to a unified one can significantly reduce the time and effort needed for implementing the consistency management framework. Furthermore, such methods can be useful for other model-based methods where unification is required.

In the definition of the concept of consistency management in Section 3, we discuss the concept of `ModelRelation`. We describe this as an abstract concept capable of representing any relationship between two models. According to our consistency management framework, introduced in Section 5, a `ConsistencyRule` is responsible for capturing the properties and characteristics of such a relationship, which are evaluated to ensure a consistent system. In our working examples (cf. Section 8), we define consistency rules for various structural relationships and dependencies. However, the limitations of this approach—specifically, the types of relationships that cannot be expressed as a `ConsistencyRule`—remain unclear. Several existing studies, such as [84, 85, 86], discuss dependencies and relationships among software components. In the context of DTs, this concept of a relationship can be extended to include dynamic relations that are only visible at runtime or the relations that exist between a RE and one or more corresponding VEs. Further research considering various notions of relationships, such as the ones mentioned above, could help identify the limitations of our approach and guide its extension to address these shortcomings.

As discussed in Section 4.1, evolution is a core characteristic of DT models [87]. The consistency management framework is designed to identify and

help address any consistency issues resulting from such changes (cf. Section 10). The assumption is that, at any point in time, there is only one version of each model. In long-running projects, where different versions and variants of a model can coexist, the consistency management framework is not equipped to support such scenarios. Future research can explore how to extend the consistency management framework to support these more complex scenarios. This requires developing new mechanisms to track and manage consistency across different branches of a model's evolution, ensuring correct consistency rules are effectively applied to each version. A key challenge is to handle the propagation of consistency issues from one version to others. This would involve designing semantic differencing algorithms that can intelligently compare different model versions and determine which consistency checks are relevant and where potential issues might manifest. The research can start with an investigation into existing version and variation management techniques used in the broader, well-known context of evolutionary MDE [88, 89]. Subsequent work can focus on adapting or tailoring these methods to meet the unique needs of DT models.

13. Conclusion

In this paper, we present our research on the management of the consistency of DT models. We initiate the discussion by presenting our conceptualization of consistency management, which allows us to establish a clear and unambiguous definition of consistency management and identify the core concepts related to it. We continue our discussion with a deeper look into the available literature, in order to understand the current perception of the different characteristics of DT models, their differences from models belonging to traditional MBSs, and the role of these differences in the context of consistency management of DT models. We find that DT models are, among other characteristics, more *heterogeneous*, highly *data-intensive*, and encounter more frequent *evolution*.

The list of characteristics indicates the importance of implementing proper consistency management for DTs to maintain their continued and correct functioning. Taking these characteristics into account, we identify three requirements for any system that aims to manage the consistency of DT models. With these requirements in mind and based on our definition of consistency management, we propose our consistency management framework and implement it in three relatively small yet highly relevant working examples. These

demonstrate the applicability of our framework and its potential in addressing the challenges related to consistency management in the context of DT models. We also show the potential of graph-based storage techniques to store heterogeneous data, which is particularly important for heterogeneous DT models. Based on the working examples, we discuss the ability of our framework to fully address all three requirements. We also discuss situations where implementing our framework might face challenges and propose possible solutions to address them. Furthermore, we present a detailed discussion outlining further research possibilities to improve our consistency management framework.

References

[1] B. R. Barricelli, E. Casiraghi, D. Fogli, A survey on digital twin: Definitions, characteristics, applications, and design implications, IEEE Access 7 (2019). doi:10.1109/ACCESS.2019.2953499.

[2] P. Baker, S. Loh, F. Weil, Model-driven engineering in a large industrial context — motorola case study, in: L. Briand, C. Williams (Eds.), Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 476–491. doi:10.1007/11557432_36.

[3] E. de Araújo Silva, E. Valentin, J. R. H. Carvalho, R. da Silva Barreto, A survey of model driven engineering in robotics, Journal of Computer Languages 62 (2021) 101021. doi:10.1016/j.cola.2020.101021.

[4] J. Davies, J. Gibbons, J. Welch, E. Crichton, Model-driven engineering of information systems: 10 years and 1000 versions, Science of Computer Programming 89 (2014) 88–104, special issue on Success Stories in Model Driven Engineering. doi:10.1016/j.scico.2013.02.002.

[5] A. Rodrigues da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Computer Languages, Systems & Structures 43 (2015) 139–155. doi:10.1016/j.cl.2015.06.001.

[6] D. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2) (2006) 25–31. doi:10.1109/MC.2006.58.

[7] H. M. Muctadir, D. A. Manrique Negrin, R. Gunasekaran, L. Cleophas, M. van den Brand, B. R. Haverkort, Current trends in digital twin

development, maintenance, and operation: an interview study, Software and Systems Modeling 23 (5) (2024) 1275–1305. doi:10.1007/s10270-024-01167-z.

[8] M. van den Brand, L. Cleophas, R. Gunasekaran, B. Haverkort, D. Negrin, H. Muctadir, Models meet data: Challenges to create virtual entities for digital twins, in: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 225–228. doi:10.1109/MODELS-C53483.2021.00039.

[9] H. Ahmad, A. Chaudhary, T. Margaria, Dsl-based interoperability and integration in the smart manufacturing digital thread, Electronic Communications of the EASST 81 (2022) 2021. doi:10.14279/TUJ.ECEASST.81.1198.

[10] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, A. Wortmann, Model-driven digital twin construction: Synthesizing the integration of cyber-physical systems with their information systems, Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2020 12 (2020) 90–101. doi:10.1145/3365438.3410941.

[11] F. Bordeleau, B. Combemale, R. Eramo, M. van den Brand, M. Wimmer, Towards model-driven digital twin engineering: Current opportunities and future challenges, in: Ö. Babur, J. Denil, B. Vogel-Heuser (Eds.), Systems Modelling and Management, Springer International Publishing, Cham, 2020, pp. 43–54. doi:10.1007/978-3-030-58167-1_4.

[12] L. Wright, S. Davidson, How to tell the difference between a model and a digital twin, Advanced Modeling and Simulation in Engineering Sciences 7 (2020) 1–13. doi:10.1186/S40323-020-00147-4.

[13] G. Walravens, H. M. Muctadir, L. Cleophas, Virtual soccer champions: A case study on artifact reuse in soccer robot digital twin construction, in: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 463–467. doi:10.1145/3550356.3561586.

[14] H. M. Muctadir, L. Cleophas, M. van den Brand, Maintaining consistency of digital twin models: Exploring the potential of graph-based approaches, in: 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2024, pp. 152–159. doi:10.1109/SEAA64295.2024.00031.

[15] M. G. Seok, W. J. Tan, W. Cai, D. Park, Digital-twin consistency checking based on observed timed events with unobservable transitions in smart manufacturing, IEEE Transactions on Industrial Informatics 19 (4) (2023) 6208–6219. doi:10.1109/TII.2022.3200598.

[16] E. Kamburjan, E. B. Johnsen, Knowledge structures over simulation units, in: 2022 Annual Modeling and Simulation Conference (ANNSIM), 2022, pp. 78–89. doi:10.23919/ANNSIM55834.2022.9859490.

[17] I. David, H. Vangheluwe, E. Syriani, Model consistency as a heuristic for eventual correctness, Journal of Computer Languages 76 (2023) 101223. doi:10.1016/j.cola.2023.101223.

[18] D. R. Dolk, B. R. Konsynski, Knowledge representation for model management systems, IEEE Transactions on Software Engineering SE-10 (6) (1984) 619–628. doi:10.1109/tse.1984.5010291.

[19] R. Balzer, Tolerating inconsistency, in: Proceedings of the 5th International Software Process Workshop, IEEE Computer Society, Los Alamitos, CA, USA, 1989, pp. 41,42. doi:10.1109/ISPW.1989.690408.

[20] R. Pascual, B. Beckert, M. Ulbrich, M. Kirsten, W. Pfeifer, Formal foundations of consistency in model-driven development, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification, Springer Nature Switzerland, Cham, 2025, pp. 178–200. doi:10.1007/978-3-031-75380-0_11.

[21] K. Feichtinger, K. Kegel, R. Pascual, U. Aßmann, B. Beckert, R. Reussner, Towards formalizing and relating different notions of consistency in cyber-physical systems engineering, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 915–919. doi:10.1145/3652620.3688565.

[22] S. Feldmann, S. J. I. Herzig, K. Kernschmidt, T. Wolfenstetter, D. Kammerl, A. Qamar, U. Lindemann, H. Krcmar, C. J. J. Paredis, B. Vogel-Heuser, A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study, in: 2015 IEEE International Conference on Automation Science and Engineering (CASE), Vol. 2015-October, IEEE, 2015, pp. 158–165. doi:10.1109/CoASE.2015.7294055.

[23] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, Inconsistency handling in multiperspective specifications, IEEE Transactions on Software Engineering 20 (8) (1994) 569–578. doi:10.1109/32.310667.

[24] B. Schatz, P. Braun, F. Huber, A. Wisspeintner, Consistency in model-based development, in: 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003. Proceedings., 2003, pp. 287–296. doi:10.1109/ECBS.2003.1194810.

[25] T. Mens, R. Van Der Straeten, J. Simmonds, A framework for managing consistency of evolving uml models, in: Software Evolution with UML and XML, IGI Global, 2005, pp. 1–30. doi:10.4018/978-1-59140-462-0.ch001.

[26] H. Giese, R. Wagner, From model transformation to incremental bidirectional model synchronization, Software & Systems Modeling 8 (1) (2008) 21–43. doi:10.1007/s10270-008-0089-9.

[27] G. Jürgen, G. Holger, S. Wilhelm, A. B. rn, F. Ursula, H. Stefan, P. Sebastian, T. Matthias, Towards the design of self-optimizing mechatronic systems: Consistency between domain-spanning and domain-specific models, Guidelines for a Decision Support Method Adapted to NPD Processes (2007).

[28] S. Biffl, D. Winkler, R. Mordinyi, S. Scheiber, G. Holl, Efficient monitoring of multi-disciplinary engineering constraints with semantic data integration in the multi-model dashboard process, in: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), IEEE, 2014, pp. 1–10. doi:10.1109/ETFA.2014.7005211.

[29] J. Rieke, R. Dorociak, O. Sudmann, J. Gausemeier, W. Schäfer, Management of cross-domain model consistency for behavioral models of

mechatronic systems, in: Proceedings of DESIGN 2012, the 12th International Design Conference, 2012, pp. 1781–1790.

[30] T. Mens, R. V. D. Straeten, M. D'Hondt, Detecting and resolving model inconsistencies using transformation dependency analysis, in: Model Driven Engineering Languages and Systems, Vol. 4199 LNCS, Springer Verlag, 2006, pp. 200–214. doi:10.1007/11880240_15.

[31] S. Feldmann, K. Kernschmidt, M. Wimmer, B. Vogel-Heuser, Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering, Journal of Systems and Software 153 (2019) 105–134. doi:10.1016/j.jss.2019.03.060.

[32] A. Egyed, Automatically detecting and tracking inconsistencies in software design models, IEEE Transactions on Software Engineering 37 (2) (2011) 188–204. doi:10.1109/TSE.2010.38.

[33] S. Feldmann, F. Hauer, D. Pantförder, F. Pankratz, G. Klinker, B. Vogel-Heuser, Management of inconsistencies in domain-spanning models – an interactive visualization approach, in: S. Yamamoto (Ed.), Human Interface and the Management of Information: Information, Knowledge and Interaction Design, Springer International Publishing, Cham, 2017, pp. 71–87. doi:10.1007/978-3-319-58521-5_5.

[34] L. Marchezan, W. K. Assuncao, E. Herac, F. Keplinger, A. Egyed, C. Lauwerys, Fulfilling Industrial Needs for Consistency Among Engineering Artifacts, Proceedings - International Conference on Software Engineering (2023) 246–257doi:10.1109/ICSE-SEIP58684.2023.00028.

[35] T. Mens, R. V. D. Straeten, Incremental resolution of model inconsistencies, in: Recent Trends in Algebraic Development Techniques, Vol. 4409 LNCS, Springer, Berlin, Heidelberg, 2007, pp. 111–126. doi:10.1007/978-3-540-71998-4_7.

[36] S. Feldmann, M. Wimmer, K. Kernschmidt, B. Vogel-Heuser, A comprehensive approach for managing inter-model inconsistencies in automated production systems engineering, IEEE International Conference on Automation Science and Engineering 2016-November (2016) 1120–1127. doi:10.1109/COASE.2016.7743530.

[37] S. Feldmann, S. J. Herzig, K. Kernschmidt, T. Wolfenstetter, D. Kammerl, A. Qamar, U. Lindemann, H. Krcmar, C. J. Paredis, B. Vogel-Heuser, Towards effective management of inconsistencies in model-based engineering of automated production systems, IFAC-PapersOnLine 48 (2015) 916–923. doi:10.1016/J.IFACOL.2015.06.200.

[38] A. Reichwein, C. J. J. Paredis, A. Canedo, P. Witschel, P. E. Stelzig, A. Votintseva, R. Wasgint, Maintaining consistency between system architecture and dynamic system models with sysml4modelica, in: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 43–48. doi:10.1145/2508443.2508451.

[39] X. Blanc, I. Mounier, A. Mougenot, T. Mens, Detecting model inconsistency through operation-based model construction, Proceedings - International Conference on Software Engineering (2008) 511–519doi:10.1145/1368088.1368158.

[40] W. Torres, M. van den Brand, A. Serebrenik, A systematic literature review of cross-domain model consistency checking by model management tools, Software and Systems Modeling (2020) 1–20doi:10.1007/s10270-020-00834-1.

[41] D. Durisic, M. Staron, M. Tichy, J. Hansson, Evolution of long-term industrial meta-models – an automotive case study of autosar, in: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 141–148. doi:10.1109/SEAA.2014.21.

[42] A. van Deursen, E. Visser, J. Warmer, Model-driven software evolution: A research agenda, in: D. Tamzalit (Ed.), Proceedings 1st International Workshop on Model-Driven Software Evolution (MoDSE), University of Nantes, 2007, pp. 41–49.

[43] D. Zowghi, C. Coulin, Requirements Elicitation: A Survey of Techniques, Approaches, and Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, Ch. 2, pp. 19–46. doi:10.1007/3-540-28244-0_2.

[44] A. Sutcliffe, N. Maiden, The domain theory for requirements engineering, IEEE Transactions on Software Engineering 24 (3) (1998) 174–196. doi:10.1109/32.667878.

[45] J. Goguen, C. Linde, Techniques for requirements elicitation, in: [1993] Proceedings of the IEEE International Symposium on Requirements Engineering, 1993, pp. 152–164. doi:10.1109/ISRE.1993.324822.

[46] J. G. Byrne, T. Barlow, Structured brainstorming: A method for collecting user requirements, Proceedings of the Human Factors and Ergonomics Society Annual Meeting 37 (5) (1993) 427–431. doi:10.1177/154193129303700507.

[47] S. R. Newrzella, D. W. Franklin, S. Haider, 5-dimension cross-industry digital twin applications model and analysis of digital twin classification terms and models, IEEE Access 9 (2021) 131306–131321. doi:10.1109/ACCESS.2021.3115055.

[48] C. L. Gargalo, S. C. de las Heras, M. N. Jones, I. Udugama, S. S. Mansouri, U. Krühne, K. V. Gernaey, Towards the development of digital twins for the bio-manufacturing industry, Advances in biochemical engineering/biotechnology 176 (2021) 1–34. doi:10.1007/10_2020_142.

[49] X. Liu, D. Jiang, B. Tao, F. Xiang, G. Jiang, Y. Sun, J. Kong, G. Li, A systematic review of digital twin about physical entities, virtual models, twin data, and applications, Advanced Engineering Informatics 55 (2023) 101876. doi:10.1016/J.AEI.2023.101876.

[50] L. Lattanzi, R. Raffaeli, M. Peruzzini, M. Pellicciari, Digital twin for smart manufacturing: a review of concepts towards a practical industrial implementation, International Journal of Computer Integrated Manufacturing 34 (2021) 567–597. doi:10.1080/0951192X.2021.1911003.

[51] M. Singh, E. Fuenmayor, E. Hinchy, Y. Qiao, N. Murray, D. Devine, Digital twin: Origin to future, Applied System Innovation 4 (2021) 36. doi:10.3390/asi4020036.

[52] J. Bao, D. Guo, J. Li, J. Zhang, The modelling and operations for the digital twin in the context of manufacturing, Enterprise Information Systems 13 (2019) 534–556. doi:10.1080/17517575.2018.1526324.

[53] C. Cronrath, L. Ekstrom, B. Lennartson, Formal properties of the digital twin-implications for learning, optimization, and control, IEEE International Conference on Automation Science and Engineering 2020-August (2020) 679–684. doi:10.1109/CASE48305.2020.9216822.

[54] F. Tao, M. Zhang, A. Nee, Five-dimension digital twin modeling and its key technologies, in: F. Tao, M. Zhang, A. Nee (Eds.), Digital Twin Driven Smart Manufacturing, Academic Press, 2019, pp. 63–81. doi:10.1016/B978-0-12-817630-6.00003-5.

[55] F. Mhenni, F. Vitolo, A. Rega, R. Plateaux, P. Hehenberger, S. Patalano, J. Y. Choley, Heterogeneous models integration for safety critical mechatronic systems and related digital twin definition: Application to a collaborative workplace for aircraft assembly, Applied Sciences 2022, Vol. 12, Page 2787 12 (2022) 2787. doi:10.3390/APP12062787.

[56] W. Jia, W. Wang, Z. Zhang, From simple digital twin to complex digital twin part i: A novel modeling method for multi-scale and multi-scenario digital twin, Advanced Engineering Informatics 53 (2022) 101706. doi:10.1016/J.AEI.2022.101706.

[57] S. Shirowzhan, S. M. E. Sepasgozar, R. Ahmad, L. Zhang, T. Ademujimi, V. Prabhu, Digital twin for training bayesian networks for fault diagnostics of manufacturing systems, Sensors 2022, Vol. 22, Page 1430 22 (2022) 1430. doi:10.3390/S22041430.

[58] H. Boyes, T. Watson, Digital twins: An analysis framework and open issues, Computers in Industry 143 (2022) 103763. doi:10.1016/j.compind.2022.103763.

[59] S. Yoon, Building digital twinning: Data, information, and models, Journal of Building Engineering 76 (2023) 107021. doi:10.1016/j.jobe.2023.107021.

[60] E. Glaessgen, D. Stargel, The digital twin paradigm for future nasa and u.s. air force vehicles, 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference;20th AIAA/ASME/AHS Adaptive Structures Conference;14th AIAA (2012) 1818doi:10.2514/6.2012-1818.

[61] D. Jones, C. Snider, A. Nassehi, J. Yon, B. Hicks, Characterising the digital twin: A systematic literature review, CIRP Journal of Manufacturing Science and Technology 29 (2020) 36–52. doi:10.1016/J.CIRPJ.2020.02.002.

[62] L. Zhang, L. Zhou, B. K. Horn, Building a right digital twin with model engineering, Journal of Manufacturing Systems 59 (2021) 151–164. doi:10.1016/j.jmsy.2021.02.009.

[63] E. Gamma, R. Helm, R. Johnson, J. Vlissides, D. Patterns, Design patterns: elements of reusable object-oriented software, Design Patterns (1995).

[64] H. Zhu, P. A. V. Hall, J. H. R. May, Software unit test coverage and adequacy, ACM Computing Surveys 29 (4) (1997) 366–427. doi:10.1145/267580.267590.

[65] I. Barosan, A. A. Basmenj, S. G. R. Chouhan, D. Manrique, Development of a virtual simulation environment and a digital twin of an autonomous driving truck for a distribution center, in: Software Architecture, Springer International Publishing, Cham, 2020, pp. 542–557. doi:10.1007/978-3-030-59155-7_39.

[66] E. Seidewitz, H. Miyashita, M. Wilson, H. P. de Koning, Z. Ujhelyi, I. Gomes, T. Schreiber, C. Bock, B. Grill, K. Zoltán, S. Marquez, W. Piers, A. Adavani, A. Graf, Sysml-v2-pilot-implementation, https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation, [Accessed April 12, 2024] (9 2023).

[67] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An evolving query language for property graphs, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1433–1445. doi:10.1145/3183713.3190657.

[68] T. Parr, The definitive ANTLR 4 reference, The Pragmatic Bookshelf, 2013.
URL http://digital.casalini.it/9781680505016

[69] S. Efftinge, M. Spoenemann, Xtext - The Grammar Language — eclipse.dev, https://eclipse.dev/Xtext/documentation/301_grammarlanguage.html, [Accessed February 2, 2024] (2024).

[70] R. Hebig, D. E. Khelladi, R. Bendraou, Approaches to co-evolution of metamodels and models: A survey, IEEE Transactions on Software Engineering 43 (2017) 396–414. doi:10.1109/TSE.2016.2610424.

[71] S. Gil, E. Kamburjan, P. Talasila, P. G. Larsen, An architecture for coupled digital twins with semantic lifting, Software and Systems Modeling (Nov 2024). doi:10.1007/s10270-024-01221-w.

[72] P. Talasila, C. Gomes, P. H. Mikkelsen, S. G. Arboleda, E. Kamburjan, P. G. Larsen, Digital twin as a service (dtaas): A platform for digital twin developers and users, in: 2023 IEEE Smart World Congress (SWC), IEEE, 2023, pp. 1–8. doi:10.1109/SWC57546.2023.10448890.

[73] T. Blockwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, A. Viel, Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models, in: Proc. 9th International Modelica Conference, Vol. 76, Linköping University Electronic Press, 2012, pp. 173–184. doi:10.3384/ecp12076173.

[74] C. Gomes, C. Thule, D. Broman, P. G. Larsen, H. Vangheluwe, Co-simulation: A survey, ACM Comput. Surv. 51 (3) (May 2018). doi:10.1145/3179993.

[75] E. Kamburjan, V. N. Klungre, R. Schlatte, E. B. Johnsen, M. Giese, Programming and debugging with semantically lifted states, in: R. Verborgh, K. Hose, H. Paulheim, P.-A. Champin, M. Maleshkova, O. Corcho, P. Ristoski, M. Alam (Eds.), The Semantic Web, Springer International Publishing, Cham, 2021, pp. 126–142. doi:10.1007/978-3-030-77385-4_8.

[76] E. Kamburjan, A. Pferscher, R. Schlatte, R. Sieve, S. L. T. Tarifa, E. B. Johnsen, Semantic reflection and digital twins: A comprehensive overview, in: M. Hinchey, B. Steffen (Eds.), The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday, Springer Nature Switzerland, Cham, 2025, pp. 129–145. doi:10.1007/978-3-031-73887-6_11.

[77] S. Jouili, V. Vansteenberghe, An empirical comparison of graph databases, in: 2013 International Conference on Social Computing, 2013, pp. 708–715. doi:10.1109/SocialCom.2013.106.

[78] D. Fernandes., J. Bernardino., Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb, in: Proceedings of the 7th International Conference on Data Science, Technology and Applications - DATA, INSTICC, SciTePress, 2018, pp. 373–380. doi:10.5220/0006910203730380.

[79] W. Ali, M. Saleem, B. Yao, A. Hogan, A.-C. N. Ngomo, A survey of rdf stores & sparql engines for querying knowledge graphs, The VLDB Journal 31 (3) (2022) 1–26. doi:10.1007/s00778-021-00711-3.

[80] A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, A. Zimmermann, Knowledge graphs, ACM Comput. Surv. 54 (4) (Jul. 2021). doi:10.1145/3447772.

[81] A. Hogan, D. Vrgoć, Querying graph databases at scale, in: Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 585–589. doi:10.1145/3626246.3654695.

[82] I. Robinson, J. Webber, E. Eifrem, Graph databases: new opportunities for connected data, O'Reilly Media, Inc., 2015.

[83] M. Shahin, M. Ali Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access 5 (2017) 3909–3943. doi:10.1109/ACCESS.2017.2685629.

[84] E. Fregnan, T. Baum, F. Palomba, A. Bacchelli, A survey on software coupling relations and tools, Information and Software Technology 107 (2019) 159–178. doi:10.1016/j.infsof.2018.11.008.

[85] D. M. German, J. M. Gonzalez-Barahona, G. Robles, A model to understand the building and running inter-dependencies of software, in: 14th Working Conference on Reverse Engineering (WCRE 2007), IEEE, 2007, pp. 140–149. doi:10.1109/WCRE.2007.5.

[86] L.-G. Yu, S. Ramaswamy, Component dependency in object-oriented software, Journal of Computer Science and Technology 22 (2007) 379–386. doi:10.1007/s11390-007-9058-y.

[87] T. Alskaif, Ö. Babur, F. Bordeleau, L. Cleophas, B. Combemale, J. Denil, Ø. Haugen, J. Michael, P. Nguyen, T. Seceleanu, M. van den Brand, H. Vangheluwe, Evolution at the core of digital twin engineering, in: 28th International Conference on Model Driven Engineering Languages and Systems (MODELS-C): EDTconf'25), IEEE, 2025, presented at the 2nd Int. Conf. on Engineering Digital Twins (EDTconf'25). doi:10.5283/epub.77656.

[88] J. G. M. Mengerink, B. van der Sanden, B. C. M. Cappers, A. Serebrenik, R. R. H. Schiffelers, M. G. J. van den Brand, Exploring dsl evolutionary patterns in practice - a study of dsl evolution in a large-scale industrial dsl repository, in: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - MODELSWARD, INSTICC, SciTePress, 2018, pp. 446–453. doi:10.5220/0006605804460453.

[89] S. Ananieva, H. Klare, E. Burger, R. Reussner, Variants and versions management for models with integrated consistency preservation, in: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 3–10. doi:10.1145/3168365.3168377.