Cognitive Digital Thread Tool-chain for Model Versioning in Model-Based Systems Engineering

Shouxuan Wu[a,d], Guoxin Wang[a,*], Jinzhi Lu[b], Jianyu Huang[a], Jiaxing Qiao[a], Yan Yan[a] and Dimitris Kiritsis[c,d]

[a]School of Mechanical Engineering, Beijing Institute of Technology, Beijing, 100081, China
[b]School of Aeronautic Science and Engineering, Beihang University, Beijing, 100083, China
[c]SCI-STI-DK, École polytechnique fédérale de Lausanne, Lausanne, 1015, Switzerland
[d]SIRIUS, Department of Informatics, University of Oslo, Oslo, 0373, Norway

ARTICLE INFO

Keywords:
Cognitive Digital Thread, Model-based Systems Engineering, Model Versioning, Open Service for LifeCycle Collaboration, KARMA language, Knowledge Graph

ABSTRACT

Model-based systems engineering (MBSE) allows system models to formalize end-to-end systems engineering implementation while developing complex engineering system. The evolution of MBSE models, including changes and conflicts, provides important historical knowledge to support design decisions. Model versioning is an efficient approach to manage the evolution of MBSE models. However, the heterogeneous data structure and semantics used in MBSE practices hinder the tool interoperability that is required in model versioning, which also decreases the effectiveness and efficiency of system development. This paper proposes a tool-chain for model versioning of MBSE models based on a cognitive digital thread (CDT). In this tool-chain, the graph-object-point-property-relationship-role-extension (GOPPRR-E) modeling approach is adopted because it is compatible with heterogeneous modeling languages used in model versioning. To promote tool interoperability, this tool-chain adopts the Open Services for Lifecycle Collaboration to support conflict detection or resolution during model versioning. In particular, knowledge graphs are generated along with the model versioning workflow to develop a CDT, which provides the cognitive reasoning ability required for model versioning behaviors. A case study of landing gear system development is used to evaluate the feasibility of the proposed tool-chain through qualitative and quantitative analyses. The results demonstrate that the proposed tool-chain has better efficiency than traditional model versioning using Git tools.

1. Introduction

Complex engineering systems face ever-increasing demands on system performance, which drives the tighter integration of engineering disciplines such as software, hardware, mechanical, and electrical engineering. This convergence in turn increases the need for a novel and holistic approach to analysis, design, and evaluation of systems at early system lifecycle stages. Model-based systems engineering (MBSE) is considered the next paradigm for designing complex systems, and it uses models to formalize end-to-end systems engineering implementation [1]. When developing complex systems by MBSE, these models developed in different modeling, simulation, and design tools are called MBSE models [2], including SysML models, Modelica models, etc. The iterative evolution of MBSE models brings about challenges related to managing consistent representation of system characteristics, which leads to the problems of providing incorrect or out-dated knowledge for design decision making. Currently, **model versioning** is already considered a key approach to manage model evolution in the MBSE domain [3]. In this paper, we focus on the **model versioning of MBSE models**. This is defined as *a process of tracking, organizing, and maintaining different versions of MBSE models along a timeline. The model versioning of MBSE models includes a set of operations on MBSE models, such as model difference, model change, and model merge*.

Currently, the model versioning of MBSE models still faces the following challenges. The semantic heterogeneous characteristics [4] in MBSE models should be first overcome. Compared with code version control in software engineering, MBSE models adopt different modeling specifications and meta-models, which include heterogeneous semantics and syntax. These heterogeneous characteristics make it difficult to capture the semantic information of the model, such as its system composition, interfaces, and parameters. This then leads to inconsistent understandings about MBSE models across stakeholders. Second, the model versioning process requires tool interoperability between MBSE tools. Different MBSE tools provide different approaches to operating data, models, and tool application programming interfaces (APIs), thereby making it difficult to achieve interoperability. When design conflicts exist in a collaborative model versioning process, poor tool interoperability results in challenges in modifying and merging design conflicts between heterogeneous models. Third, MBSE model versioning must account for specifying, detecting, and resolving the addition architecture complexity produced by design changes. For example, changes in system requirements will bring about synchronous changes in system compositions, parameters, and interfaces, which may exist across domain and design tools. A lack of mechanisms to handle these changes increases the design costs of system development.

*Corresponding author; Email: wangguoxin@bit.edu.cn
ORCID(s):

The digital thread (DT) [5] is regarded as a framework for information management and integration at all stages of the product lifecycle for decision making. The DT can serve as a theoretical support for MBSE model versioning. Recently, semantic technologies have been used as key enabling components in many intelligent systems to achieve semantic interoperability for heterogeneous data and information [6]. Semantic models, such as ontology and knowledge graphs, use intuitive, concise, and unified semantic topological interrelationships to generate new knowledge for MBSE models by employing a reasoner [7]. In previous studies [8, 9], we proposed a concept of the **cognitive digital thread** (CDT),which serves as a DT with enhanced semantic ability that allows it to formalize traceability relationships between system artifacts, development processes, and stakeholders. Based on the concept of CDT, we have designed a service-oriented MBSE tool-chain prototype to correlate heterogeneous design information and for reasoning and decision making in several traceability scenarios. However, due to the ineffective management of model changes across different versions, the collaborative efficiency of the tool-chain is hindered. In this paper, we decide to extend the CDT concept to model versioning scenarios of MBSE models.

The main contribution of this article is **a CDT-based tool-chain for model versioning of MBSE models**. In this tool-chain, we first develop a set of MBSE models for model-based design and verification. These MBSE models, which formalize system architecture on appropriate abstraction, are developed by the same meta-meta models so that they share model semantics. Secondly, based on Open Service for Lifecycle Collaboration (OSLC) specification, this tool-chain provides web-based unified services to integrate heterogeneous models and APIs of multi-domain tools to improve design automation. Such unified services support information exchange and tool operations in model versioning workflow, such as modifying a conflict property when merging MBSE models. Finally, this tool-chain utilizes knowledge graphs to identify the complex topological interrelationships in a model versioning process, such as recording version changes relationships between model repositories, branches, commits, and conflicts. Furthermore, knowledge graphs that link the URLs of OSLC services of MBSE models are embedded into the model versioning workflow. These steps formalize tacit design knowledge in the model evolution process, thereby building a DT for stakeholders to understand the complex dependencies between MBSE models. In addition, these knowledge graphs enhance the cognitive capabilities of the DT by analyzing changes in MBSE models across different versions using predefined reasoning rules. This promotes the comprehension and prediction of model evolution during the model versioning of MBSE models.

The rest of this paper is organized as follows. We discuss the related work in Section 2 and present our research methodology in Section 3. In Section 4, we introduce our proposed tool-chain in detail. In Section 5, a case study of the model versioning process for a landing gear system (LGS) design is shown in order to illustrate how the tool-chain works. Section 6 evaluates our approach through the case study and compares it with a traditional model versioning process. Finally, we offer conclusions in Section 7. The glossary for this article is shown in Table 1.

Table 1
Glossary along with the definition

Acronym	Definition
MBSE	Model-Based Systems Engineering
API	Application Programming Interface
DT	Digital Thread
CDT	Cognitive Digital Thread
OSLC	Open Service for Lifecycle Collaboration
LGS	Landing Gear System
VCS	Version Control System
UML	Unified Modeling Language
BPMN	Business Process Modeling Notation
OML	Ontological Modeling Language
GOPPRR-E	Graph, Object, Point, Property, Relationship, Role, and Extension
SysML	System modeling language
MC	Mouse Click
MM	Mouse Movement
LOC	Line of Code
URL	uniform resource locator
RE	Requirement Engineer
SE	System Engineer
LE	Landing Gear System Engineer
RD	Requirement Diagram
UC	Usecase Diagram
BDD	Block Definition Diagram
IBD	Internal Block Diagram
ACT	Activity Diagram
PAR	Parameter Diagram
SMD	Simulink Diagram

2. Related Works

In this section, we first review the current literature regarding the model versioning of MBSE models, including related concepts, approaches, and tools. Then, we discuss the concept of CDT and current efforts to create a CDT for model versioning in MBSE practice. Finally, we analyze the challenges faced by MBSE model versioning.

2.1. Model versioning

For almost six decades, version control systems (VCSs) have been an indispensable part of the collaborative tools in software development. With the development paradigm of software shifting toward model-driven engineering, model versioning is considered an important issue for the configuration management of models throughout the software life cycle [3]. Altmanninger et al. [10] conducted a survey and presented an overview of the top-level features of VCSs for model versioning; these features included collaborating, merging, repository architecture, and branching. Several approaches and their implementations have been proposed

to support model versioning in model-driven engineering, and examples include AMOR [11] and EMF Compare [12]. These works mainly concentrate their focus on specific modeling languages, especially languages based on eclipse modeling framework meta-models, such as Unified Modeling Language (UML) and Business Process Modeling Notation (BPMN). The supports of model versioning for other emerging modeling languages, such as object-process language, have been ignored, which hinder the scalability of these works.

MBSE, which supports the engineering and development efforts [13] of complex systems by formalizing development processes, system architectures, and operational interrelationships, has been widely applied in various industrial sectors. Recent MBSE practices encourage more integration of multiple modeling languages instead of using only one specific language. For example, models developed by system modeling language (SysML) and architecture analysis and design language were integrated to support the development of flight control system [14] and fault management of avionics software [15]. At present, the influence of abstract syntax and concrete syntax on model versioning has been paid more attention in the MBSE domain. For instance, Exelmans et al.[16] proposed an operation-based approach for the optimistic versioning of models in abstract syntax and concrete syntax during collaborative hybrid modeling. The diversity of languages used in MBSE practice brings about potential challenges in building an adaptable and language-specific framework for model versioning. This is especially true from the perspectives of model comparison, conflict detection, and user support for the conflict resolution of MBSE models.

2.2. Tool-chains supporting model versioning in MBSE

The MBSE tool-chain is defined as *one tool-chain consisting of two or more modeling, simulation, or design tools that, when combined, can support and construct a system engineering workflow* [2]. Supports for model versioning are available in many integrated MBSE tool-chain environments. Open Model-Based Engineering Environment (OpenMBEE), which serves as an integrated collaborative environment and MBSE tool-chains led by NASA Jet Propulsion Laboratory, provides several techniques for model versioning for SysML models, including a model management system [17] and Lemontree [18]. The model management system is a basic part of the OpenMBEE platform and stores SysML model data from Magicdraw. This data includes classes, instances, attributes, values, relationships, and their change histories. Lemontree can be integrated into OpenMBEE by using a patched MDK of Magicdraw to provide three-way model difference and model merge operations for SysML models. Another of NASA's initiatives is OpenCAESAR (Computer Aided Engineering for Systems Architecture) [19]. OpenCAESAR provides a Git-based versioning workflow combined with vocabulary in ontology modeling language (OML) to improve the integrated electrical and harness engineering processes for Europa Clipper [20]. Modelio Constellations, which includes VCS by Apache Subversion, provides versioning and configuration management abilities that allow modelers to work on shared architecture models based on SysML and UML for contributing architecture deliverables in MegaM@Rt2 projects [21].

Various existing commercial tools, such as MagicDraw, Enterprise Architect, and MetaEdit+, provide support for MBSE model versioning. Rocco et al. [22] provided an overview of some of these tools and discussed their version control capability for specific models like UML and SysML models. MagicDraw enables teamwork for the pessimistic version control of models (i.e., lock-modify-unlock). Enterprise Architect provides XMI-package-based methods for model versioning as well as interfacing with third-party VCSs. Meanwhile, MetaEdit+ provides model comparison and integration with external VCSs like Git and TortoiseSVN to support model versioning for MBSE models [23]. However, without an open and unified standard for tool integration, the time and cost must be large for efforts realizing data interoperability to manipulate heterogeneous models in MBSE tool-chains.

2.3. Cognitive digital thread for model versioning in MBSE

The DT was first proposed for F-35 fighter jet development [24]. As digital engineering strategies [25] evolve, the DT concept is constantly being updated and expanded upon. West et al. [26] considered it as "a framework for merging the conceptual models of the system with the discipline-specific engineering models of various system elements." Kraft et al. [5] considered it as "an extensible, configurable, and enterprise level framework that seamlessly expedites the controlled interplay of authoritative data, information, and knowledge to inform decisions during a system lifecycle by providing the capability to access, integrate, and transform disparate data into actionable information." The DT maintains information, such as versions and contextualizing links, about digital models of the system, thus enabling stakeholders to reduce mistakes and rework related to incorrect models. Several platforms have been designed for version management of digital models, such as CAD models in ship design [27] and UML and CAD models in virtual aircraft engine design [28]. However, it remains a great challenge to clarify the semantics of digital models and their inter-relationships in a DT.

Recently, semantic technologies like ontology and knowledge graphs have been applied in DTs to formalize digital models and their inter-relationships. Ontology has been successfully used as semantic web frameworks for checking and inter-operating SysML models in unmanned aircraft system design [29], and it has been employed in service orchestration in co-simulation scenarios of an MBSE tool-chain [30]. Knowledge graphs provide semantic inter-relationship

representations for digital models in distributed manufacturing systems [31] and for semantic mechanical standards like STEP, EXPRESS, and QIF at manufacturing and inspection phases [32] [33]. The category theory, which provides a mathematical language to describe the relationships between different versions of a model, also formalizes conflict detection and management in collaborative scenarios concerning the electro-mechanical actuators of ailerons [34]. In addition, some researchers have attempted to enhance the DT with cognitive capabilities using semantic technologies. The paradigm of the next-generation DT was proposed as CDT [8]. The CDT serves as a DT with enhanced semantic ability to formalize the interrelationship between the model and data, development processes, and organizations. Researchers have put more emphasis on semantic representations for traceability relationships between MBSE models from the perspectives of various disciplines, yet very few works in semantic representations have addressed version evolution relationships from a timeline perspective.

2.4. Summary

Based on literature reviews, we can summarize several key challenges and corresponding contributions related to our work. (1) **Semantic heterogeneity**: As seen from Section 2.1, to support model versioning of MBSE models by different modeling languages to describe domain-specific perspectives, the model versioning framework should be adaptive to, or at least configurable to, these modeling languages. (2) **Tool interoperability**: From Section 2.2, we can see that although the aforementioned tool-chains and commercial tools support model versioning of MBSE models by tool-specific approaches, the tool integration under an integrated development environment should be taken into account, particularly when realizing model versioning of MBSE models. (3) **Model evolution analysis**: As discussed in Section 2.3, to enhance the cognitive ability and address deficiencies in knowledge representation and reasoning in model versioning over time, the designed tool-chain should utilize semantic technologies for analyzing the version evolution of models.

3. Research Design

The research design of this paper is depicted in Figure.1. We combine the problem solving and case study methods to design our tool-chain and evaluate its prototype.

3.1. Problem solving

The problem solving is a method to obtain the solution by detailing the problem [35]. In order to obtain the tool-chain solution, we analyze some major concerns in model versioning detailly. First, by identifying the problems and challenges in building a model versioning tool-chain in MBSE, three main challenges are captured in this paper: challenges in semantic heterogeneity, tool interoperability, and model evolution analysis for the model versioning. The diversity among MBSE models in domains, model semantics, and model syntax makes it hard to define the minimal granularity of model elements for model versioning. The practice of MBSE entails the interaction of different tools, which means it is difficult to balance the development cost, redundancy, and flexibility of tool interfaces when integrating heterogeneous tools in a model versioning workflow. Besides, the information generated during model versioning, such as version changes and conflict-solving strategies, contains the tacit design knowledge about model evolution over a timeline. Enhancing analysis of these information may be valuable for future automation of MBSE model versioning.

Second, to solve the identified challenges, we look to corresponding technical solutions and strategies. To overcome semantic heterogeneity, unified MBSE modeling languages and tool integration techniques are adopted. Unified MBSE modeling languages provide standardized meta-models to provide integrated architecture representation during model versioning, which eliminates the semantic inconsistency between MBSE models. Tool integration techniques facilitate model information exchange during model versioning by developing tool interfaces. To promote tool interoperability during model versioning, version control service, file system/repository, and database techniques are taken into consideration in addition to the necessary tool integration techniques. Version control services (i.e., conflict identification and solving services) using a service-oriented approach to interoperate models and tool APIs within the tool-chain, which provides a loose-coupling and flexible way to manipulate tools. To fully store different versions of MBSE models, it is necessary to provide a storage and indexing mechanism for the model files, which requires the involvement of database techniques and file system/repository techniques. The requirement for analyzing model evolution of MBSE models can be satisfied by adopting version control workflow, version control services, graph-based/rule-based analysis, and database techniques. Version control workflow and services provide mechanisms to update MBSE models in a collaborative way. Database techniques and graph-based/rule-based analysis techniques are used to store and analyze the implementation information (such as modified items in a specific version change).

Finally, to implement these designed technical solutions, some key technologies are selected. We select four key technologies to design our tool-chain. To solve the problems caused by the use of heterogeneous languages during model versioning, KARMA language [36] are adopted to formalize the system architecture because of its strong compatibility with heterogeneous meta-models. To support tool integration and provide version control services, an MBSE tool adapter is developed to provide OSLC-compliant services [37] as middlewares to operating MBSE models and tool APIs during model versioning. Given that even for the SysML, which is the most common used modeling language in MBSE domain, there is currently no open-source MBSE model versioning engine tailored to this language, we decided to developed our own model versioning engine to provide model versioning workflow and services. To record version history information for model evolution analysis,

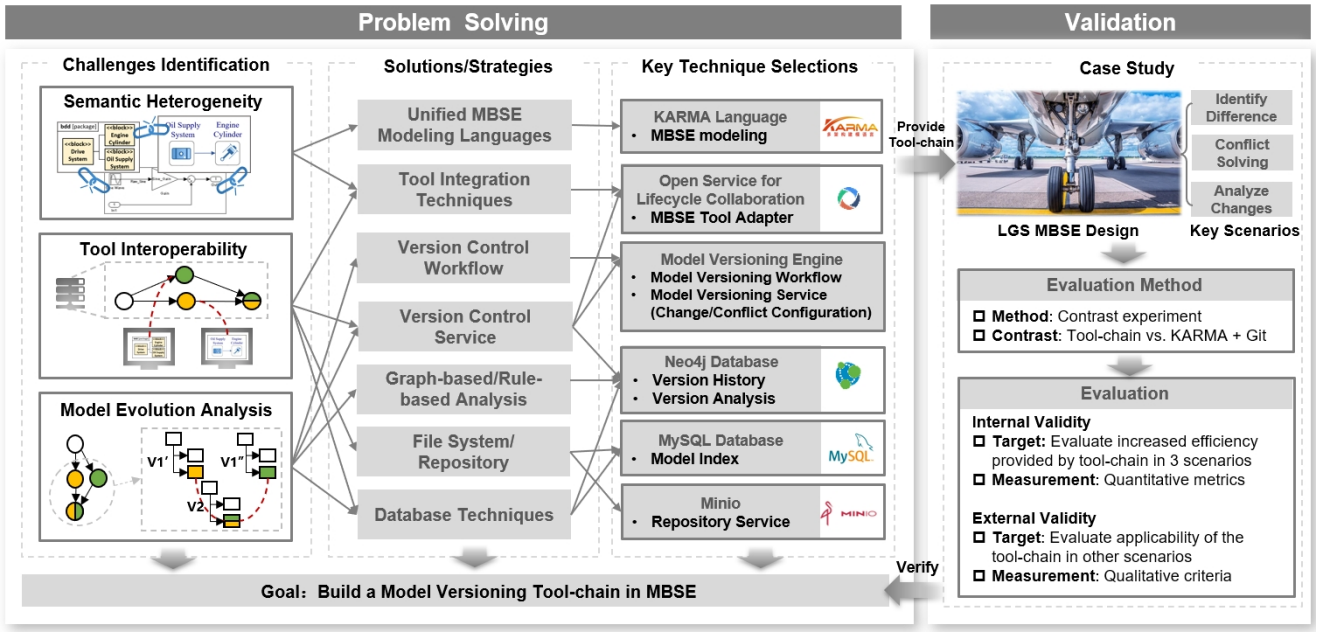

Figure 1: Overview of the research design

the Neo4j Database, which includes nodes embedded with OSLC services, is adopted as graphic database storage and analysis tools. In addition, MySQL database together with the Minio tool provides required services for model index and model repository, which serve as file system/repository for model versioning.

3.2. Case-study-based evaluation

Our tool-chain prototype embodies the main contribution of this article. The case study method is an effective way to evaluate its effectiveness [38]. In this paper, a collaborative landing gear system (LGS) development by MBSE models is adopted as a case study for evaluating the proposed tool-chain. This case study contains several key scenarios of model versioning, including identifying model differences, solving conflicts, analyzing model changes. Internal validity and external validity are two important aspects of a case study approach [39]. Therefore, internal and external validity are used to evaluate the contribution of this paper.

In this paper, the internal validity of the case study refers to **the increased efficiency due to the use of the proposed tool-chain in a model versioning process**. We adopt methods of quantitative evaluation proposed by Wang et al. [40] and Lu et al. [41] to design a quantitative contrast experiment to evaluate the developed tool-chain. The KARMA language adopted in the proposed tool-chain provides both graphical model view and textual code view, aligning with the current trends in advanced modeling languages such as SysML 2.0 and object-process language. These languages can be seamlessly integrate with mature code version control technologies and tools (e.g., Git). Each line of these KARMA codes independently represents the design information in the model. Thus, changes of KARMA code will be directly reflected as the changes of objects, properties and other features in different versions of models. Since the KARMA language is adopted in this paper as the unified underlying modeling language, the most appropriate tool-chain as comparison is OpenCAESAR because it utilizes OML as its unified foundational language within their tool-chain. By establishing the mapping mechanism between OML models and OML code, they make use of Git for model versioning of OML models. However, a direct comparison between OpenCAESAR and the proposed tool-chain in this study is not feasible, as there are no available universal datasets or test cases that can be applied simultaneously to both tool-chains, given their differing technology stacks. For the aforementioned reasons, in this paper, we have chosen the Gitee (`https://gitee.com`), a code version management tool and workflow based on Git, as a comparison approach for our tool-chain prototype to manage MBSE models, evaluating them both qualitatively and quantitatively. The experiment can be broken down into the following three parts.

(1) *Selecting participant*: Before this experiment, the experiment participant with background of SysML and Simulink modeling had received a one-month training on how to use the modeling tool *MetaGraph* to develop MBSE models, and the basic concepts of model versioning. Then this participant are taught to use the developed tool-chain and Gitee as two solutions for model versioning tasks.

(2) *Specifying the problem and task*: In this case study, we specify the task of model versioning as **developing MBSE models of the LGS from the same initial model and merging these models**. The initial MBSE models of the LGS are provided. The demands and

tasks about how to merge two given MBSE models are explained clearly to the participant.

(3) *Selecting the assessment criteria*: The efficiency of model versioning is assessed by the numbers of mouse click (MC), mouse movement (MM), length of input code(LOC), and consumed time (T) during model versioning. The result of these four metrics are captured by a mouse record tool[1]. The MC refers to the number of mouse clicks when designers implement related model versioning operations. The MM refers to the length of movement when designers use their mouse to implement related operations. The LOC refers to lines of code which designers need to input during model versioning. The T refers to the total time consumed during model versioning process. Thus, if the tool-chain shows lower measurement results of the aforementioned metrics than those of the Git-based approach, it indicates that the efficiency of model versioning of the proposed tool-chain is better than the Git-based approach.

In our study, the external validity of the case study refers to **the extent to which the proposed tool-chain can be replicated in other model versioning scenarios**. After comprehensively considering three key challenges (i.e., semantic heterogeneity, tool interoperability, and model evolution analysis) in model versioning, we decide to adopt the qualitative criteria proposed by Altmanninger et al. [10] to evaluate the scalability of the proposed tool-chain. The qualitative criteria are listed below.

(1) *Domain-specific flexibility* refers to the capability that the tool-chain adapts to heterogeneous modeling languages and meta-models. It can be measured by the domain-specific perspectives represented by the models.
(2) *Capability of operation detection* refers to the capability that the user operations on models are identified and recorded. It is determined by the minimum granularity of models that can be identified in the model versioning workflow.
(3) *Capability of conflict detection* refers to how well contradictory design behaviors on models are identified. This can be measured by assessing the patterns of design conflicts in the model versioning workflow.
(4) *Capability of conflict resolution* refers to how well models can be accessed and manipulated to solve design conflicts. It can be measured by understanding the tool interoperability within the tool-chain.

4. Cognitive digital thread tool-chain supporting model versioning in MBSE

In this section, we first provide an overview of the proposed tool chain. This is followed by an introduction to MBSE modeling for model-based design and verification. Next, we discuss the service-oriented approach for conflict detection and recording changes. Finally, we introduce the service-embedded knowledge graph for model versioning.

[1] Available: https://www.mouserecorder.com/index.html

4.1. Overview of the tool-chain

Figure 2 shows the overview of the designed tool-chain. This tool-chain are composed of three major parts, including a modeling tool *MetaGraph*, a Java-based OSLC tool adapter, and a web-based cognitive digital thread platform. When the tool-chain is implemented, the system developer first designs the MBSE meta-models and then builds MBSE models in the *graphs, objects, points, properties, roles, and relationships with extensions* (GOPPRR-E) modeling environment provided by *MetaGraph*. Then, utilizing the Java-based compiler, we develop an OSLC tool adapter for *MetaGraph*. Through this OSLC adapter, the information of MBSE models are parsed by a model parser. Then, concepts of MBSE model (such as graphs or objects) are transformed into OSLC core model concepts and to generate corresponding OSLC services by service generator of the OSLC adapter. These OSLC services are used for manipulating MBSE models and tool APIs of *MetaGraph* by the model versioning workflow in the CDT platform.

In previous research [9], we have developed a web-based CDT platform that contains Neo4j database and OSLC adapters to establish and analyze traceability relationships betweeen MBSE models in Excel, *MetaGraph*, and Simulink. Based on the existing CDT platform, we continued to develop a Java-based model versioning engine. This engine provide a branch-based model versioning workflow for parallel development of MBSE model, together with a set of model versioning services as basic operations (i.e., update model, create and merge branch, etc.). These model versioning services can be reused in model versioning plugins that developed in MetaGraph through REST APIs. By using the model versioning plug-in in the MetaGraph or directly upload/download MBSE models, the upload/download model versioning services utilize the database interface to persist these models to the model repository. This model repository are composed by MySQL database and Minio file system, where MySQL provides a set of model index tables while the Minio provides model storage services. The developed OSLC services are involved in the model versioning workflow by REST APIs in the following three parts: ① identify how model change between different versions; ② detect design conflicts of MBSE models with the same source; and ③ solve design conflicts of MBSE models with the same source. The model versioning engine provides dynamic model versioning mechanism to update MBSE models. In addtion, the behaviors of MBSE model between versions (i.e., add, delete, modify, and no change) and update items are persisted in the Neo4j database as the version history. By using the Cypher languages in the Neo4j database, we can imitate human intuition and cognitive thinking mode to construct direct and multi-hop reasoning template for cognitive reasoning of model evolution. This reasoning ability can also be reused in model versioning plugins of MetaGraph through REST APIs.

The tool-chain is detailed below from three aspects: (1) GOPPRR-E methodology for MBSE modeling; (2) the

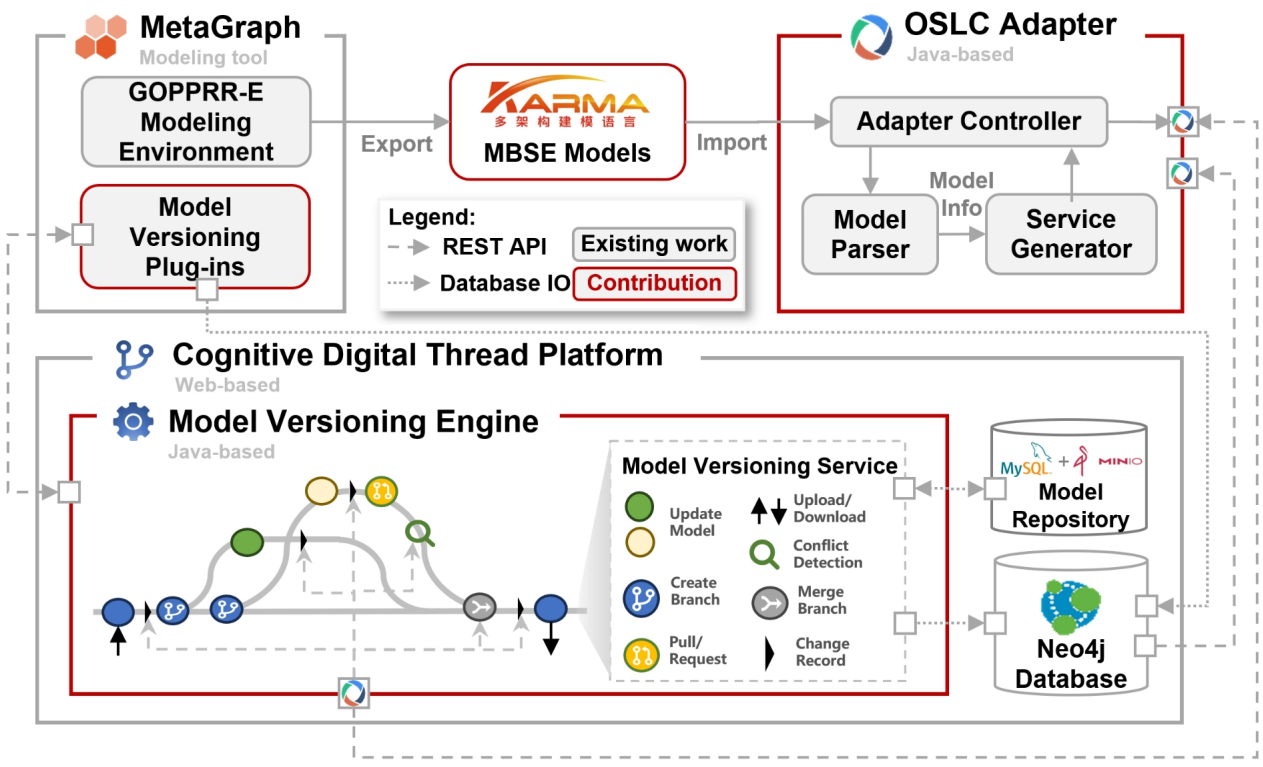

Figure 2: Overview of the proposed tool-chain

service-oriented approach supporting model versioning; and (3) cognitive knowledge graph construction.

4.2. GOPPRR-E methodology for MBSE modeling

The model versioning of MBSE models should first handle the semantic heterogeneity challenge brought about by using different modeling languages to develop MBSE models. This heterogeneity stems from the differing underlying implementation mechanisms of various modeling languages. For instance, both SysML and Modelica are commonly used object-oriented modeling languages in the MBSE domain. However, SysML is a graphical modeling language based on the MOF mechanism, emphasizing system representation without support for solving; whereas Modelica is a declarative modeling language based on differential, algebraic, and discrete equations, with language features that can support non-causal modeling and compile-time solving. The characteristics of modeling languages are often reflected in the design of their metamodels. Therefore, the ability of model versioning for MBSE models with custom metamodels is a crucial issue that needs to be addressed. To solve this problem, we use the KARMA language present in the GOPPRR-E methodology [36] to describe domain-specific characteristics. Comparing to existing meta-modeling languages and methodlogies (e.g.,ARIS, Ecore, GME, GOPPRR, and Microsoft Visio), the GOPPRR-E modeling methodology has the most excellent descriptive capabilities formalizing possible relationships patterns of models [42]. Considering the applicability of the proposed tool-chain, we do not give a set of specific meta-models. Instead, we provide a method for constructing MBSE models based on the GOPPRR-E methodology and use a landing scenario to aid in the explanation. The process of MBSE modeling based on GOPPRR-E methodology is shown in Figure 3. The MBSE modeling process can be divided into four levels based on the top-down principle.

1) The M0 level is a set of GOPPRR-E meta-meta-models used to define how to design meta-models. The GOPPRR-E meta-meta-models are described below.
   a) **G**raph refers to the domain viewpoint of a system using a symbolic representation of the model's topology that includes objects, properties, points, roles, relationships, and extensions. The example includes the *Internal Block Diagram (IBD)* in SysML.
   b) **O**bject refers to an entity that constructs a graph, representing basic system elements. For example, there is the *part property* object in the *IBD* .
   c) **P**roperty refers to an attribute of another meta-model. For example, there are *name, type* properties belonging to the *part property* object in the *IBD*.
   d) **P**oint refers to the port with connectivity on an object. Examples are the *proxy port* points on the *part property* object in the *IBD*.
   e) **R**elationship refers to the connection between objects, and it can be used to describe the system's topological structure. Here we use the *connector interact* relationship to express the power/signals transfer relationship between *part property* objects.

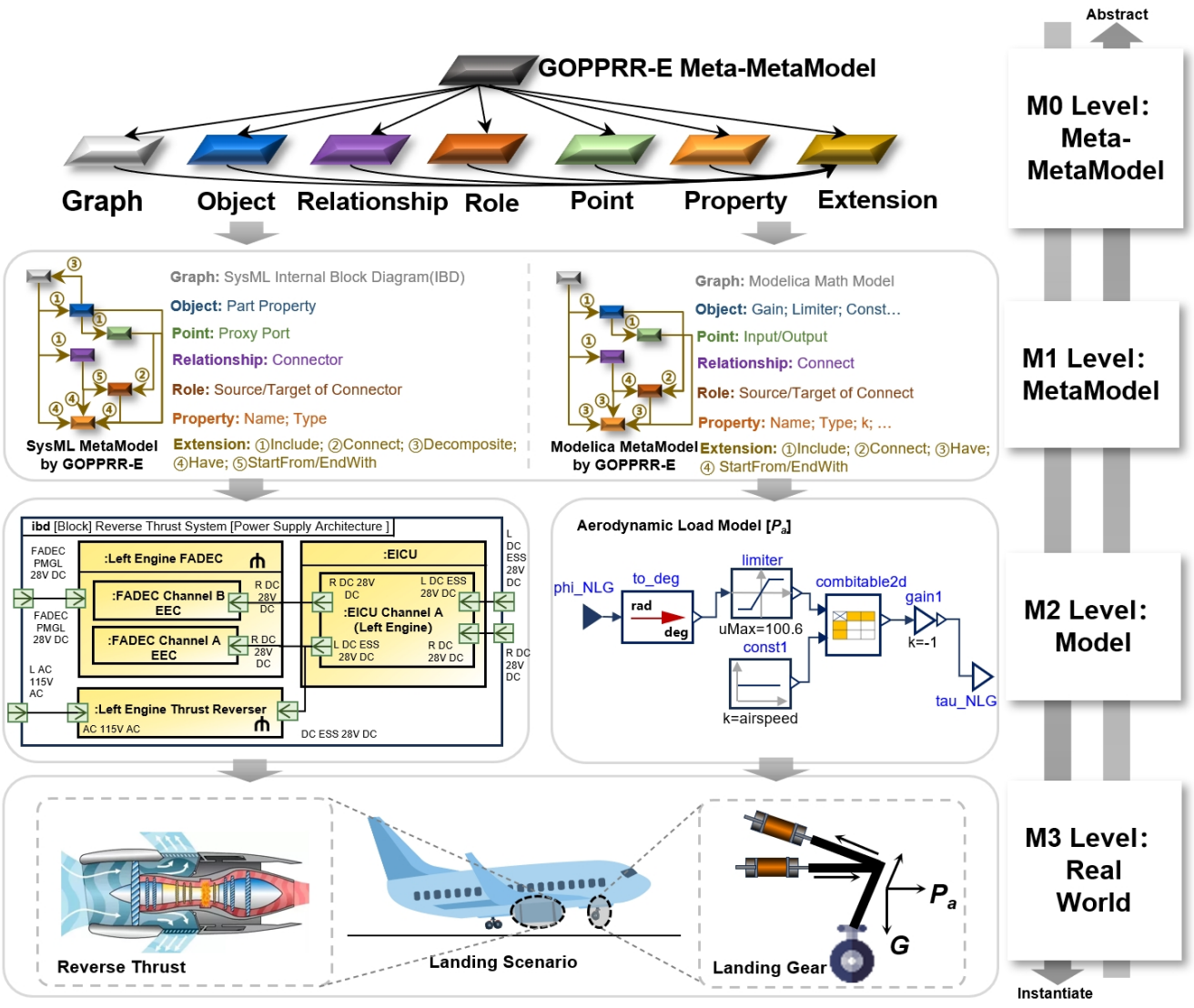

Figure 3: MBSE modeling based on the GOPPRR-E methodology

f) **R**ole refers to the appendant of a relationship, and it can identify the direction of the relationship. For example, we use *source/target of the connector* roles to express the beginning and end of the *connector* relationship.

g) **E**xtension refers to the additional constraints used to construct meta-models, including the association relationships between meta-models. Here, we have five types of extension relationships in SysML meta-models: ① Include (i.e., the *IBD* graph includes the *part property* object and the *connector* relationship; the *part property* object includes the *proxy port* point); ② Connect (i.e., the *proxy port* point connects the *source/target of the connector* roles); ③ Decomposite (i.e., the *part property* object can be decomposited as an *IBD* graph); ④ Have (i.e., the *IBD* graph, the *part property* object, the *proxy port* point, and the *connector* relationship both have *name and type* properties); ⑤ StartFrom/EndWith (i.e., the *connector* starts from or ends with *source/target of the connector* roles).

2) The M1 level includes a set of metamodels that can be reused to design models with specific characteristics. Here in Figure 3, we used the GOPPRR-E meta-metamodels to develop both meta-models of SysML *IBD* and Modelica math models, which means that these metamodels share consistent underlying modeling concepts. For example, we use the concept of objects to respectively represent the *part property* block in the *IBD* and the *gain* component for multiplication calculation in Modelica. Within this context, considering the language specifications of SysML and Modelica, the object concept of *part property* includes a point concept of *proxy ports*, as well as property concepts such as *name* and *type*. The object concept of *gain* includes point concepts of the *input* and *output*, as well as property

concepts such as *name* and *K* (rate of multiplication). The extension relationship between these objects and points is *include*, while the extension relationship between objects and attributes is *have*. Since they share same underlying modeling concepts, the semantic heterogeneity, which is caused by adopting the different modeling specifications, between these SysML metamodels and Modelica metamodels can be eliminated to the greatest extent.

3) The M2 level includes a set of models that extract and abstract the key elements in real-world scenarios. For example, the power supply and control architecture composed by the full authority digital electrical control (FADEC) system, electrical interface control unit (EICU), and thrust reverser can be represented using *IBD*. The nested *part properties* block express the composition of systems and components, while the transmission of power/signals is modeled using a combination of *proxy ports* and *connectors* in the *IBD*. During the retraction process of the LGS, the performance of the LGS are influenced by aerodynamic loads, which can be modeled using the Math model by the Modelica modeling language. During the retraction process, the magnitude and direction of the aerodynamic load are influenced by the flight speed and the angle of landing gear retraction. Therefore, the aerodynamic load model is developed based on a two-dimensional linear interpolation table in the Modelica language. The data within this interpolation table represents aerodynamic load values at different flight speeds and various landing gear retraction angles. The model takes the landing gear angle as input and provides the corresponding torque as output.
4) The M3 level represents the real world and includes real physical scenarios and physical entities involved in the development process. For example, the landing scenario requires the involvement of two key systems: the thrust reverse system and the LGS. The thrust reverse system generates additional braking force by altering the direction of the aircraft's engine exhaust, allowing the aircraft to decelerate rapidly. During thrust reversal, engine and power control for the thrust reverse system are regulated based on signals such as the aircraft's runway distance. Meanwhile, the retracting and deploying process of the LGS is primarily influenced by its own weight, aerodynamic resistance, and actuator cylinder thrust/tension.

By using the GOPPRR-E methodology in our tool-chain, we aim to expand the scope of model versioning at the metamodel level of the MBSE models, addressing the challenge of semantic heterogeneity in model versioning. The MBSE models and their metamodels relevant to the designed tool-chain are all implemented using the GOPPRR-E methodology. Additionally, based on the GOPPRR-E methodology and the KARMA language, we can develop MBSE models based on other modeling languages and specifications such as UML, BPMN, and EAST-ADL, etc [36, 43], which also ensures the scalability of the developed tool-chain.

4.3. OSLC-based approach for tool interoperation in model versioning

To support tool interoperation during model versioning, we develop the OSLC tool adapter to transform MBSE models into OSLC services. Afterward, these OSLC services are integrated into the model versioning workflow to serialize models for identifying design changes or conflicts between MBSE models in different versions.

4.3.1. Generating OSLC services for MBSE models

In the proposed MBSE tool-chain, the OSLC services of MBSE models are generated by developing an OSLC tool adapter in Java. The OSLC tool adapter mainly includes a model parser and a services generator. Figure 4 explains how OSLC adapter works to transform MBSE models into OSLC services. Although the metamodels of MBSE models is variable, they are all developed based on the same GOPPRR-E meta-metamodels. Similar to the relationship between models and metamodels, firstly, we need to develop OSLC concept models of MBSE models based on the OSLC core specifications [37] before developing OSLC services. In Figure 4, the OSLC concept models are developed by establishing the mapping rules in the model parser of the OSLC adapter. In this paper, we focus on the mapping rules between MBSE models adopting GOPPRR-E modeling methodology and OSLC services. While in previous studies, we introduced the mapping rules between other MBSE models (e.g., SysML, Simulink, Modelica, etc.) and OSLC concept models [9, 44, 41]. We use the example of the *IBD* to illustrate how MBSE models and the API for manipulating them are mapped to concepts in the OSLC concept models:

a) An *Resource* is an atomic object structure that is configured to be consumed by the OSLC adapter. For example, the instances of graph (*IBD*), object (*part property*), point (*proxy port*), properties (*name* and *type*), relationship (*connector*), and roles (*source/target of the connector*) metamodels can be consider OSLC *Resources*. Each resource is identified by a uniform resource locator (URL) and has the same attribute structure with MBSE models, such as *name* and *type* properties in a *proxy port* point.
b) A *Service* is a set of capabilities that enable a web client to manage *Resources* through REST APIs. In MBSE models, the APIs for operating model instances are considered *Services*. Such APIs include those for creating, querying, updating, and deleting the instances of GOPPRR metamodels. A *Service* offers information for viewing the link to a *Resource* and rich previews of the *Resource*.
c) A *ServiceProvider* is a container that provides an implementation of *Services*. In MBSE models, the concepts of graph/objects/properties/points/relationships/roles metamodels can be considered *ServiceProviders*. A *ServiceProvider* lists links of *Services*.
d) A *ServiceProviderCatalog* is a list used in the discovery of OSLC *ServiceProviders*. The concept of a MBSE model can be considered a *ServiceProviderCatalog*.

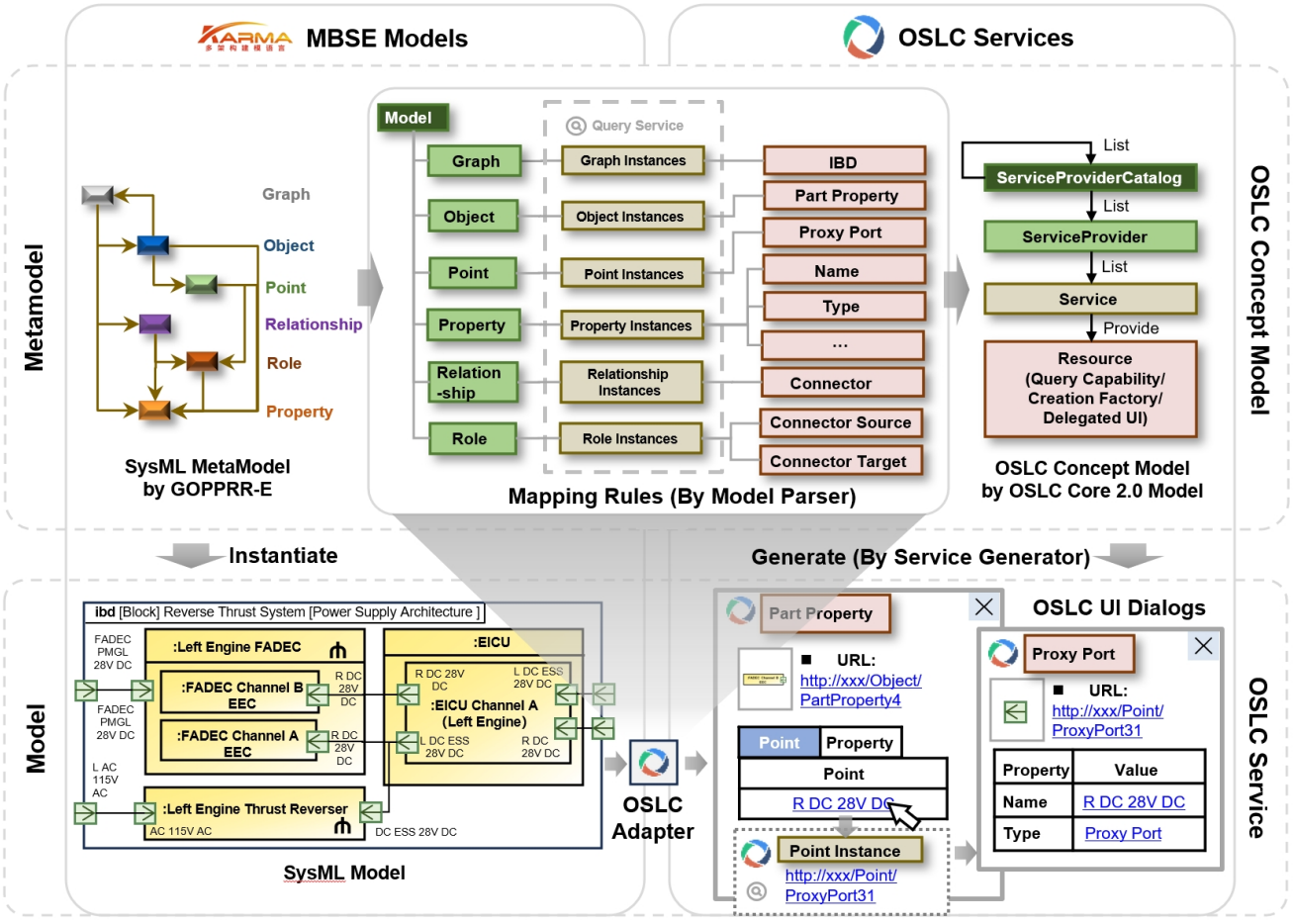

Figure 4: Mapping between OSLC services and MBSE models

After developing these OSLC concept models, the OSLC adapter will generate OSLC service automatically through its service generator. The service generator works in two parts, including automatically generating a service discovery chain and generating the delegated UI of OSLC service. Figure 4 shows an example of generated OSLC services. In the MBSE model, there is a input *proxy port* point named R DC 28V DC on the *part property* named FADEC Channel B EEC. This relationship is automatically transformed into a service discovery chain between OSLC *Resources* UI dialogs of the *part property* and *proxy port*. By clicking the URL of the aforementioned point instance in the UI dialog of the *part property* FADEC Channel B EEC, the *Service* for querying point instance is invoked through REST API, linking to the corresponding UI dialog of the *proxy port* and shows all its properties information. Through the web-based OSLC service generated by the OSLC adapter, the model information and its operation APIs can be further accessed and invoked during the model versioning process to support version-related operations.

4.3.2. *Integrating OSLC services into model versioning engine*

After OSLC services are generated, they are integrated into the model versioning workflow to improve tool interoperability in model versioning operations. In this paper, the model versioning workflow and its related operation services are implemented based on the model versioning engine. Figure 5 illustrates the execution process, artifacts, and components about this engine. The model versioning workflow is designed to be branch-based, which is similar to Git [45] and allows different users to make independent changes to the same model at the same time on their respective branches. There are three core components in the model versioning engine, including an OSLC service parser, conflict/change detection algorithms, and a model versioning service provider. The OSLC service are linked to the engine by an OSLC services parser. Because OSLC services serialize model information into hierarchical and interrelated service information, the OSLC services parser provides an algorithm to for traversing all the information through the URL of the OSLC *ServiceProviderCatalog* of an MBSE model, which is shown in function *Discovery* in

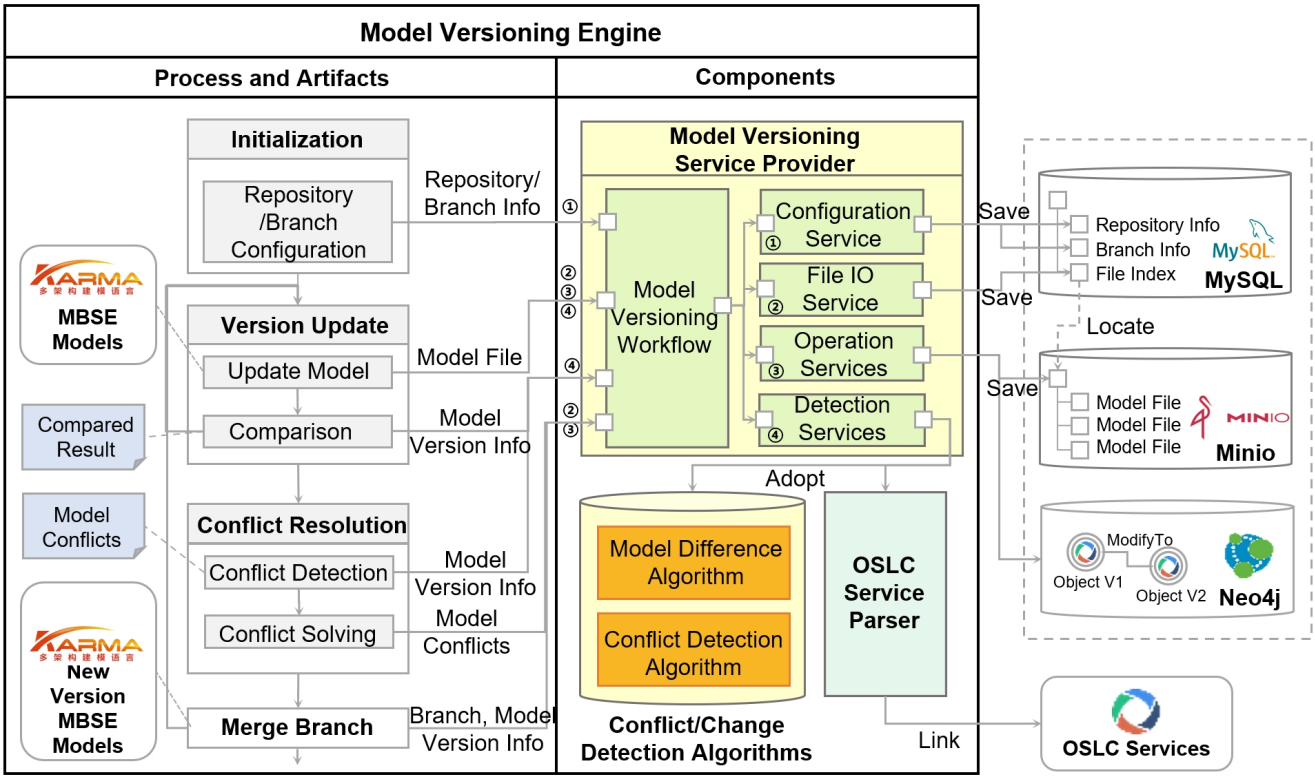

Figure 5: Model versioning engine

algorithm 1. Currently, the conflict/change detection algorithms include two types of algorithms, which are the model difference algorithm (algorithm 2) and the conflict detection algorithm (algorithm 3). These two algorithms both use the OSLC URL as unique identification of model elements, and use this URL as the ID of model difference and conflict detection for ID-based comparison.

Algorithm 1 Service Discovery for OSLC services

Input: *URL*, Given URL of OSLC services
Output: *List*, Discovered list of OSLC services
function DISCOVERY(*URL*) **begin**
  List = ∅;
  result = HttpRequest(*URL*);
  if *result* has *ServiceProviders* **then**
    for *ServiceProvider* from *ServiceProviders* **do**
      Discovery(URL in ServiceProvider);
  else if *result* has *Services* **then**
    for *Service* from *Services* **do**
      Discovery(URL in Service);
  else if *result* has *Resources* **then**
    List = *List* ∪ *Resources*;
    return *List*;
  end if
end function

The third core component of the model versioning engine is the model versioning service provider. Each task

Algorithm 2 Model Difference by OSLC services

Input: *List,List'*, OSLC services list for two MBSE models
Output: *AddList*, Added elements list in MBSE models
Output: *ModList*, Modified elements list in MBSE models
Output: *DelList*, Deleted elements list in MBSE models
function CHANGE(*List,List'*) **begin**
  AddList = *List*; *DelList* = *List'*; *ModList*= ∅;
  for *Resource* from *List* **do**
    for *Resource'* from *List'* **do**
      if *Resource'.id* = *ResourceIni.id* **then**
        if *Resource'* != *ResourceIni* **then**
          ModList.add(*Resource'*);
        end if
        AddList.remove(*Resource'*);
        DelList.remove(*Resource*);
      end if
    end for
  end for
  return [*AddList*,*ModList*,*DelList*];
end function

and operation in the model versioning process is scheduled by the model versioning workflow service of the model versioning service provider component in this engine to associate and reuse the corresponding model versioning service. There are four types of model versioning services: ① **Configuration service**. The configuration service can

Algorithm 3 Conflict Detection by OSLC services

Input: *URL*,*URL'*,*URL"*, URLs for initial MBSE model file and two MBSE model files extend the initial MBSE model
Output: *Add'*, *Mod'*, *Del'*,*Add'*, *Mod'*, *Del'*,*Conf'*, List of design changes and conflicts in MBSE models

```
function DETECT(iniHash,exHash',exHash") begin
    [List,List',List"]=DISCOVERY([URL,URL',URL"]);
    Add'=Add"=Mod'=Mod"= ∅;
    Del'=Del"=Conf= ∅;
    [Add',Mod',Del'] = CHANGE([List,List'])
    [Add",Mod",Del"] = CHANGE([List,List"])
    for Resource' from Mod' do
        for Resource" from Mod" do
            if Resource'.id = Resource".id and Resource'
!= Resource" then
                Conf.add([Resource',Resource"]);
            end if
        end for
    end for
    return [ Add', Mod', Del',Add', Mod', Del',Conf'];
end function
```

extract, store, and modify the model repository information and branch information in the MySQL database; ② **File IO service**. The file IO service persists the model file into Minio, and establishes the actual storage location, version description, and other metadata and index information of the model file in the MySQL database; ③ **Operation service**. The operation service supports dynamic modification of model files for tasks such as design conflict resolution, as well as storing model version information and version update relationships in Neo4j database (detailed introduced in Section 4.4); ④ **Detection service**. The detection service combine the OSLC services and conflict/change detection algorithms to identify model changes/conflicts.

The model versioning process begins with the initialization task, including the user configuration for the repositories and branches of MBSE models. The model repository and branch information generated by this task is stored in MySQL database through the configuration service. Based on these information, repository and branch instances of the model version workflow are dynamically generated. After the initialization task is complete, the MBSE model can be uploaded to the CDT platform for the version update task. This task includes updating the MBSE model and comparing different versions of the model. When updating models, the model file is automatically indexed by the file IO service and uploaded to Minio by the operation service. In this task, the MBSE model can only be operated by one user and no conflict exists, which means they are parsed by OSLC services and model difference algorithm to obtain the model update information (including update items and update operations) that is further stored in the Neo4j database through the operation service. When comparing different versions of MBSE models, the information of these models are provided to identify and highlight changes between them using the model difference algorithm (algorithm 2). After discovering and linking OSLC services in different levels through algorithm 1, the algorithm 2 requires two discovered OSLC services lists for MBSE models as inputs and takes added/deleted/modified model elements as outputs. The OSLC resources with the same URL will be matched and compared to identify the added/deleted/modified components in MBSE models through the function *Change* in algorithm 2. The version update task is repeated several times until the MBSE model is about to be merged.

Once the model is ready to be merged, the model versioning engine enters the conflict resolution task to handle any design conflicts that may occur, including conflict detection and conflict solving. Similar with comparing different versions of MBSE models in conflict detection, the information of these models are provided to identify and highlight model conflicts between them using the conflict detection algorithm (algorithm 3) through detection services. The difference is that the input of the algorithm 3 consists of OSLC service lists generated by three versions of the MBSE model, including two different versions of the model (e.g., $V_{2.1}$ and $V_{2.2}$) and its common source version (e.g., V_1). On the basis of these identified model changes in algorithm 2, the function *Detect* in algorithm 3 further identifies whether the model changes in models $V_{2.1}$ and $V_{2.2}$ conflict with the source model V_1 by comparing the changes in OSLC services. If model conflicts exist, the model versioning engine provides the conflict solving mechanism through the operation service. The operation service links to corresponding UI dialog of OSLC service according to the ID of the model element that is marked as a conflict status. The OSLC UI dialog provides highlights of the identified conflicts in the model and APIs to modified them. After solving these conflicts, it will go the the final merge branch task. The model files and information about the branch and model version will be update through the file IO service and the operation services. To ensure the future scalability and compatibility of the tool-chain, all components of the model versioning engine are modularized developed in Java.

4.4. Knowledge graph for model evolution analysis

To support model evolution analysis during model versioning, the knowledge graph (KG) for recording the model versioning history with corresponding reasoning rules are developed in Neo4j database. As the model version evolves, instances of the KG are generated simultaneously. Before its instances are generated, the patterns of the KG should be first designed. Some important concepts in model versioning process serve as patterns of nodes and edges in these KGs, which are listed in Table 2. Given their ability to describe both model graphs (e.g., SysML BDD) and model elements (e.g., SysML block), the OSLC services are embedded in the KG as *OSLC* nodes to enhance the fidelity of model information. This reduces the size of the KG to some extent and makes it easier to analyze the model versioning history. Besides, the concepts of *model repository*, *model file*, and *branch* correspond to model repositories composed of

Table 2
Patterns in knowledge graphs

Pattern	Type	Property	Description
Node	OSLC	$< name, url, updateTime, type, hierarchy, version, committer, message, createTime, creator, parentRepository, parentFile, parentBranch, isAdded, isDeleted >$	OSLC services for model files or model components (i.e., objects, relationships, properties). In this paper, the alternative value of *type* includes $KARMA$, while alternative values of *hierarchy* include $Graph, Object, relationship, point$, and $property$.
	ModelRepository	$< name, description, createTime, creator >$	A place storing MBSE models and associated histories.
	ModelFile	$< name, path, size, createTime, creator, parentRepository >$	The model file based on KARMA language. The process saving the current state of models to the model repository with an explanatory message and unique hash ID.
	ModelBranch	$< name, description, createTime, creator, parentRepository >$	A separate or extended line for model versioning; may originate from other branches.
Edge	ModelFileInclusion	$< name, type >$	A relationship between a *ModelRepository* node and a *ModelFile* node. The alternative value of *type* includes *hasFile*.
	BranchInclusion	$< name, type >$	A relationship between a *ModelFile* node and a *ModelBranch* node. The alternative value of *type* includes *hasBranch*.
	OslcLink	$< name, type >$	The linking relationship between a *ModelBranch* node and *OSLC* node of an MBSE model file. The alternative value of *type* includes *links*.
	VersionOperation	$< name, type >$	A relationship between different *OSLC* nodes. Alternative values of *type* include *ModifyTo*, *Add*, *Delete*, *Remain*.
	MergeRelationship	$< name, type >$	A relationship between different *OSLC* nodes of MBSE model files. The alternative value of *type* includes *MergeFrom*.

MySQL and Minio, KARMA-based MBSE model files, and different branches of an MBSE model file in the tool-chain, respectively. The edges representing relationships between these concepts are illustrated in Table 2 in detail.

Figure 6 shows details about the generated KG using an example of merging two MBSE models. A merge process is depicted in the left solid frame of Figure 6. There exists a SysML block definition diagram (*BDD*) in the MBSE model file *LGS.metag* that is stored in model repository *repository A*. The details of this merge process are shown in the solid frame at the bottom of Figure 6. The initial version (V_1) of this *BDD* formalize the composition of the ATA32 brake system, which includes two SysML *blocks* representing a braking control master unit (BCMU) and a braking control valve (BCV). The working pressure of the BCV is 21 MPa. To develop this model in parallel, a new model branch *branch B* is created from current model branch *branch A*. In version $V_{2.1}$ of the *branch A*, the working pressure of the BCV is adjusted to 23 Mpa and an alternative braking control unit (ABCU) *block* is added. In version $V_{2.2}$ of the *branch B*, the working pressure of the BCV is adjusted to 22 Mpa, which forms a design conflicts with that in version $V_{2.1}$. Version V_3 represents the final design decision about the value of the BCV's working pressure and the addtion of the ABCU, which is a consensus among the designers.

The details of the generated KG for the aforementioned merging process are shown in the right solid frame of Figure 6. According to the pattern of the KG defined in Table 2 (i.e., the dotted box in the upper left corner of Figure 6), the generated KG mainly includes three sub-KGs:

1) **The sub-KG for configuration** (shown as purple cluster in the KG) represents the configuration process and relationships of model repositories, model files, and model branches. In detail, the *ModelRepository* node links to the *ModelFile* node by the *ModelFileInclusion* edge with *hasFile* type. The *ModelFile* node links to two *ModelBranch* nodes by two *BranchInclusion* edges with type *hasBranch*.
2) **The sub-KG for models** (shown as blue cluster in the KG) represents the evolution between models in different versions. It uses *OSLC* nodes to formalize MBSE models. For example, since the model *ATA 32* only changes internal model elements between different versions without adding a new model or deleting existing model, the relationships among versions V_1, $V_{2.1}$, $V_{2.2}$, and V_3 are recognized as changes by the model difference algorithm. Thus, there are four *VersionOperation* edges with type *ModifyTo* between corresponding *OSLC* nodes of the model *ATA 32* in different versions. Since the operation of merging branches happens between versions $V_{2.1}$ and

$V_{2.2}$ of the model *ATA 32*, there is a *VersionOperation* edge with type *MergeFrom* between *OSLC* nodes of *ATA32 V2.1* and *ATA32 V2.2*.

3) **The sub-KG for model elements** (shown as blue cluster in the KG) represents the evolution between GOPPRR model instances in different versions. It uses *OSLC* nodes to formalize model elements. For example, compared with version V_1, the model difference algorithm identifies the modification of the working pressure property in the *block* object *BCV* and the addition of the *block* object *ABCU* in version $V_{2.1}$. This modification is represented as adding an *OSLC* node *BCV V2.1* and establishing a *VersionOperation* edge with type *ModifyTo* with the *OSLC* node *BCV V1* in the KG; while this addition is represented as adding an *OSLC* node *ABCU V2.1*. Similar to merge relationships between *OSLC* nodes of different versions of models, the *VersionOperation* edge with type *MergeFrom* between *OSLC* nodes of *BCV V2.1* and *BCV V2.2* are synchronously built when a merge operation occurs.
4) **The relationships between three sub-KGs** include the relationships between *ModelBranch* nodes, *OSLC* nodes of models, and *OSLC* nodes of model elements in different versions. The ownership relationship between model versions and their corresponding model branches are represented by *OSLCLink* edges *links* between a *ModelBranch* node and *OSLC* node of the model. Changes to model elements in the corresponding model are represented by *VersionOperation* edges between *OSLC* nodes of model and its model elements.

After developing these KGs to capture the model versioning history, a rule-based reasoning is conducted to analyze the model versioning behaviors. A rule repository with several reasoning rules is developed based on the syntax of the Cypher query language. The Cypher language is a graph query language standard being developed by ISO, which was originally developed for graph manipulation in property graphs of the Neo4j database. The most important part of its syntax, the Cypher language provides several patterns to specify the KG. In these patterns, round brackets specify nodes of the KG (e.g., (n1:nodeLabel)), while square brackets declare edges of the KG (e.g., ()-[e1:edgeLabel]-()). For both nodes and edges, the fields before and after the colon operator in brackets specify the instance name and type of the node/edge. Besides, the nodes and edges may bear property sets, which are specified in curly brackets (e.g. (n1:nodeLabel propertyName: ”propertyValue”). By reusing and combining these syntax of the Cypher language, the reasoning pattern statement of model version history in the KG can be constructed flexibly.

By analogy with human intuition and cognition [46], direct matching rules and multi-hop matching rules are developed as reasoning rules to support cognitive reasoning of the KG. The direct matching rule corresponds to the intuitive thinking mode of humans when facing simple problems, describing the direct connections between nodes. Consequently, based on common sense information (i.e., obvious relationships between two nodes), it can rapidly make judgments within a KG. On the other hand, the multi-hop matching rule corresponds to the cognitive thinking mode of humans when considering complex problems involving multiple factors. By combining several direct matching rules to develop multi-hop matching rules, it progressively retrieves information and solves problems within the KG (i.e., conducting deeper searches to eventually uncover complex paths of relationships between nodes), leading to problem resolution. Using these rules, the model versioning operation chains are identified to provide insight about model evolution through pattern matching in Neo4j database. There are totally five types of matching rules as following:

1) *Inclusion rules*: refers to query all *ModelFileInclusion* and *BranchInclusion* edges. For example, *Match (n1: ModelRepository{name: "Repository A"})-[r: ModelFile Inclusion]->(n2:ModelFile) Return n1,r,n2* and *Match (n1:ModelFile{name:"LGS.metag"})-[r:BranchInclusion]->(n2:ModelBranch) Return n1,r,n2*.
2) *Link rules*: refers to query all *OslcLink* edges. For example, *Match (n1:ModelFile{name:"LGS.metag"})-[]->(:ModelBranch)-[r:OslcLink]->(n2:OSLC{type:"file"}) Return n1,r,n2*.
3) *Operation rules*: refers to query all *VersionOperation* edges. For example, *Match (n1:OSLC{name:"ATA32"; version:"1"; type:"model"})-[r:VersionOperation]->(n2: OSLC) Return n1,r,n2*.
4) *Merge rules*: refers to query all *MergeRelationship* edges. For example, *Match (n1:OSLC{name:"ATA32"; version:"2.1"; type:"model"})-[r:MergeRelationship]->(n2: OSLC) Return n1,r,n2*.
5) *Multi-hop matching rules*: refers to the combination of above four types of rules. For example, if a designer wants to know the information about final version V_3 of the *BCV Block* object in the *LGS.metag* as shown in Figure 6, a multi-hop matching rule can be designed as follows: *Match (n1:ModelRepository)-[:ModelFileInclusion]->(n2:ModelFile{name:"LGS. metag"}) With n2 Match (n2)-[:BranchInclusion]->(n3: Branch) With n3 Match (n3)-[:OslcLink]->(n4:OSLC {version:"3"}) With n4 Match (n4)-[:VersionOperation]->(n5:OSLC{name:"BCV"}) Return n5*.

5. Case Study

5.1. Problem analysis

To evaluate the effectiveness of our tool-chain, a model versioning process of MBSE models related to the landing deceleration scenario of the LGS is proposed as a case study. The LGS is widely used in the taking-off and landing processes of an aircraft. To ensure that our tool-chain can be used for MBSE models developed by different modeling languages, we did not adopt all the modeling elements in the SysML specification. Instead, we tailored them and combined some features of Simulink models to develop a

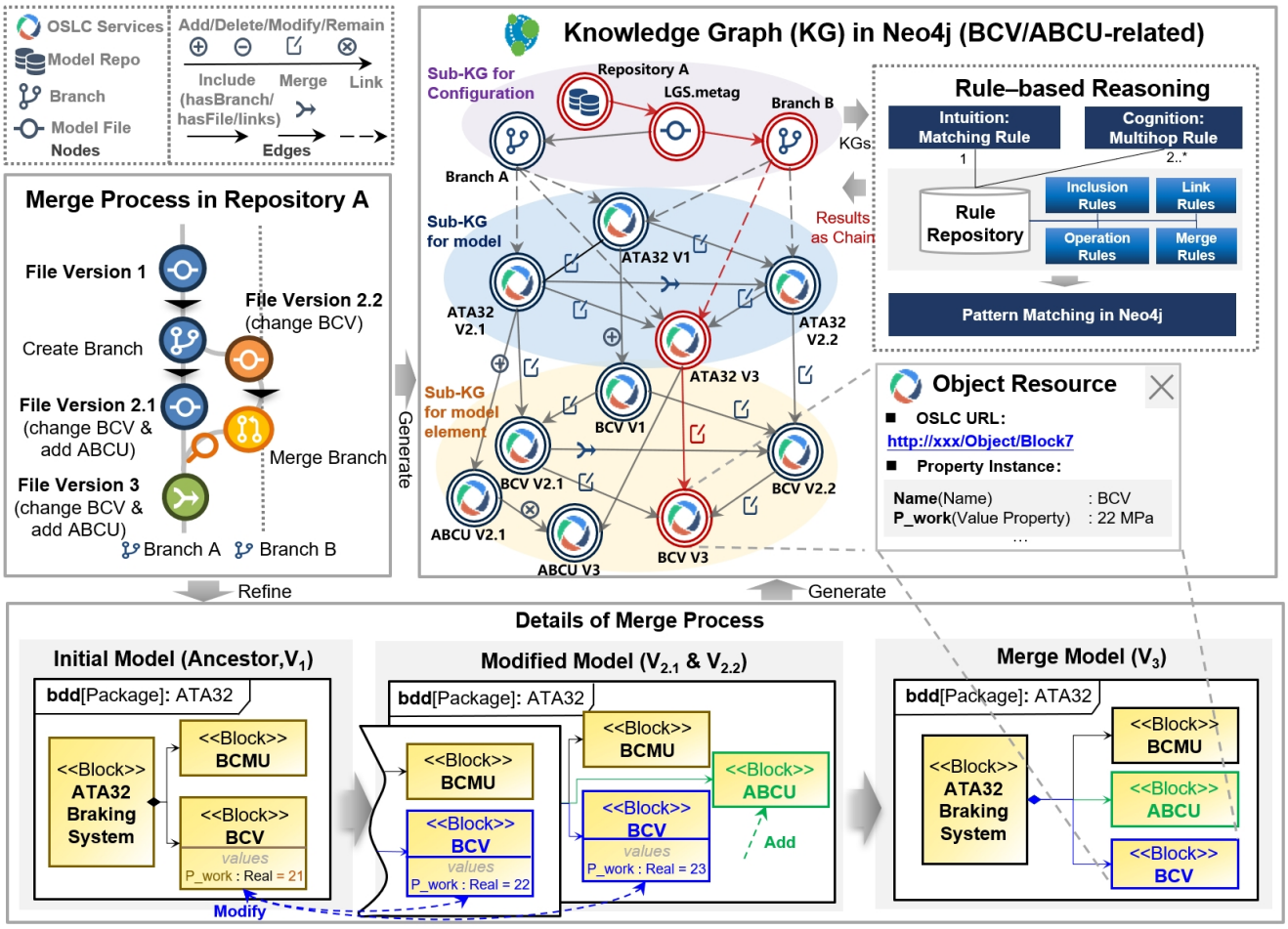

Figure 6: Knowledge graph for model versioning in Neo4j

set of hybrid metamodels based on GOPPRR-E methodology, as shown in Table 3. In the case study, the SysML graph metamodels include requirement diagram(RD), use-case diagram(UC), block definition diagram(BDD), internal block diagram(IBD), activity diagram(ACT), and parameter diagram(PAR). The corresponding object metamodels and relationship metamodels in these graph metamodels share same graphical semantic aligning with the SysML language specification version 1.6. The Simulink graph metamodels include simulink diagram (SMD). Since the detailed design for LGS models is not the main focus of this paper, only the meta-models used in this case study are listed in Table 3 and the detail design about the constraint relationships between these meta-models will not be discussed in this paper [2].

Using these metamodels, a collaborative model versioning workflow involved by three designers is defined to illustrate how our tool-chain works, which is shown in Figure 7. This model versioning workflow adopts a iterative modeling method to specifying the system and its subsystems, including following four basic stages: (1) Requirement: analyzing and capturing requirements, (2) Function: defining system structure and functions, (3) Logic: analyzing system logical behaviors, and (4) Physic: analyzing system physical performance. After these four stages, the design of the LGS will be gradually refined from initial requirements to a more detailed architecture solution. Although there may be more updates for model versions in practical applications, we found that this workflow covers all the steps needed for branch-based model versioning, and can be extend to any model versioning workflow by increasing the number of version updates in the workflow or combining two or more workflows.

Figure 8 shows details about LGS models mentioned in Figure 7. In Figure 7, we identify the types and amounts of models that the designers added and changed in green and orange during each version update. The system engineer(SE) first setup model repositories and branches to properly organize the top-level collaborative plan for final models. After that, the requirement engineer(RE) first collects stakeholder need and define preliminary system requirements using the RD in model V1 of the requirement stage. Then the system requirements are further refined by using context analysis

[2]All the resources (e.g., models, results, videos) related to this case study have been open sourced on https://gitee.com/zkhoneycomb_wushx/cdt-tool-chain-for-model-versioning-in-mbse and https://www.bilibili.com/video/BV16FweeZEeZ/?vd_source=0eeaaf8c44af71ea00cedc4c27622906

Table 3
Metamodels of LGS models

Language	Graph	Object(Property/Properties)[Decomposite]	Relationship
SysML	Requirement Diagram (RD)	Requirement(Id, Text) TestCase(Name)[ACT]	Contain Derive Refine Satisfy Verify Copy Trace
	Usecase Diagram (UC)	Actor(Name) Use Case(Name)[ACT] Block(Name, Part, Value, Port, Constraint)[IBD]	Include Extend Association Generalization
	Block Definition Diagram (BDD)	Block(Name, Part, Value, Port, Constraint)[IBD]	Composition Association Aggregation Generalization
	Internal Block Diagram (IBD)	Value Property(Name, Type, Default Value) Part Property(Name)[IBD,PAR]	Connector Item flow
	Activity Diagram (ACT)	Action(Name) Call Behavior Action(Name)[ACT] Initial Node(Name) Activity Final(Name) Flow Final(Name) Decision(Name) Merge(Name) Fork(Name) Join(Name)	Control Flow Object Flow
	Parameter Diagram (PAR)	Part Property(Name)[IBD,PAR] Value Property(Name, Type, Default Value) Constraint Property(Name, Constraint)	Connector
Simulink	Simulink Diagram(SMD)	Gain(Gain,Multiplication,SampleTime) Constant(Value,VectorParams1D,SampleTime) Display(Format,Decimation,Floating) Clock(DisplayTime,Decimation) Product(Inputs,Multiplication,CollapseMode,CollapseDim,SampleTime) Sum(IconShape,Inputs,CollapseMode,CollapseDim,SampleTime) Scope	Association

and usecase definition approach in model V2 of the function stage. Based on the system requirement, the SE can define the system-level structure, interfaces, and behaviors in the model V3. In the logic stage, the SE further refines the sub-system level architectural elements related to load distribution such as the structure, interfaces, parameters, and functional behaviors by BDD, IBD, and ACT in model V4. Meanwhile, a LGS engineer(LE) refines the system interaction of braking system by IBD and then conducts parametric modeling related to braking analysis using empirical formulas based on PAR in model V5. These changes during the logical stage both affect the system-level model structure to form design conflicts. In the physic stage, the SE resolve the design conflict in model V6. Based on this model, the LE further uses SMD to develop performance analysis model in model V6 for further generating corresponding Simulink models automatically.

In Section 3.2, we specify the core task for MBSE model versioning as **developing MBSE models of the LGS from the same initial model and then merging these models**. These tasks are subdivided into three sub-tasks, which represent three most typical scenarios in model versioning as they are shown in Figure 7: ① identifying model differences between two model versions, ② resolving design conflicts between two models in different branches and initial model, and ③ analyzing the changes between the MBSE model versions. In the remainder of this section we will discuss each of these three scenarios.

5.2. Scenario 1: Identify model difference

During each version update, the tool-chain first identifies changes between that version model and its previous version.

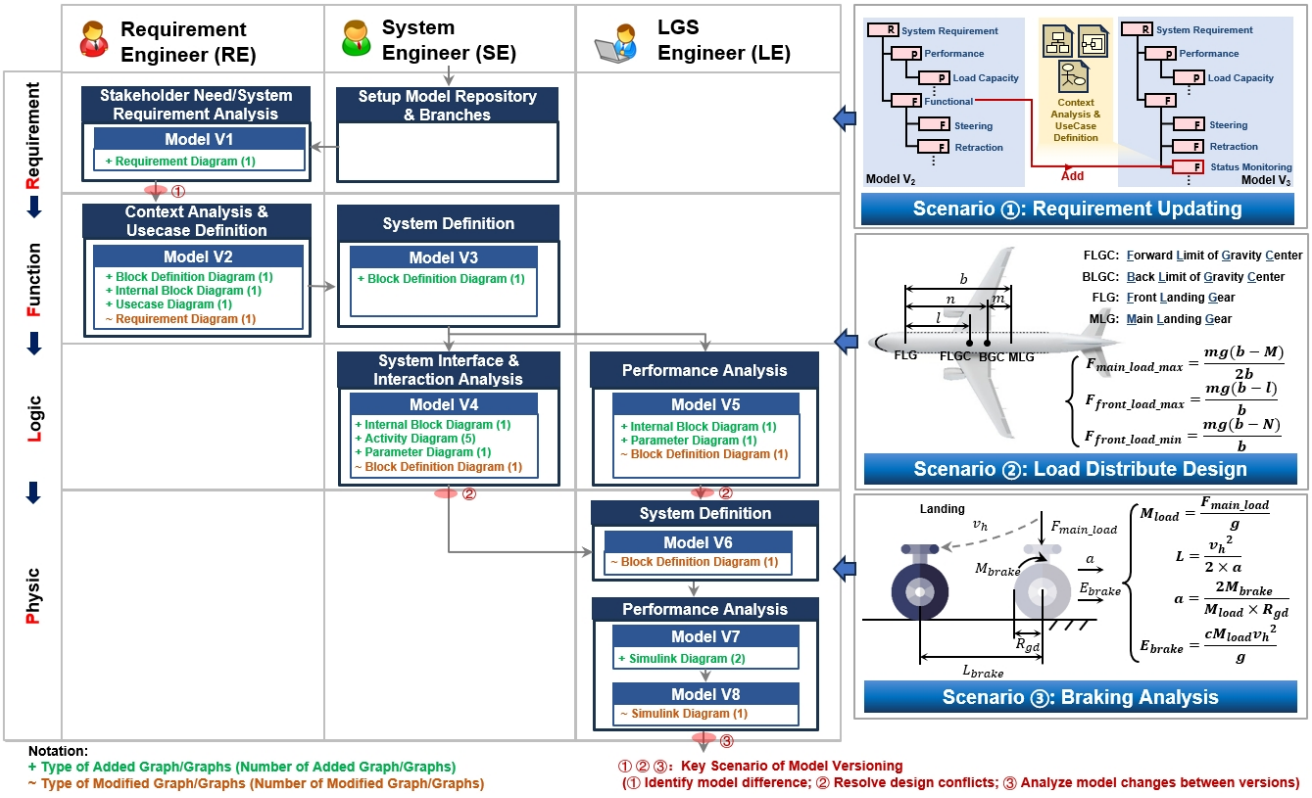

Figure 7: Model versioning workflow for LGS collaborative design

Figure 9 describe how the tool-chain identifies model differences through a scenario of changing requirements.

As shown in Figure 9(a), an RD was developed by the RE in the model V2 to capture functional and performance requirements for the LGS. Specifically, two performance requirement blocks were defined, including load capacity and braking capability. Additionally, six functional requirement blocks were established, encompassing steering, deployment, shock absorption, retraction, ground taxiing, and braking. Based on the BDD and IBD shown in Figure 9, the RE further captures the additional functional requirements of emergency retraction and deployment and status monitoring, adding them as requirement blocks in the model V3 and connecting them with the existing functional requirement block through inclusion relationships. Besides, the RE also modified the deployment requirement according to his understanding of rethinking the real scenarios. The aforementioned updates by the RE are shown in the brown dotted area of Figure 9(b).

In this scenario, since only the RE operates on the model, there are design changes between different versions rather than design conflicts. Notably, these design changes include the newly added BDD and IBD in Model V3, as well as two newly created requirements in the RD and their inclusion relationships with functional requirement. As illustrated in Figure 9(c), the RE separately uploads the MBSE model to the CDT platform. Based on the name of the model file, the CDT platform automatically selects the current model branch and generates a new version of the model under that branch. By selecting two different version of the same model file, differences between these two versions are automatically identified and highlighted. The comparison results between model V2 and V3 are shown in Figure 9(d). The information and changes within the models are visualized through delegated UI provided by the OSLC service. The model-level changes, including adding three diagrams and modifying RD, are highlighted in red text in left-side tree structures. More specific updates about model elements, such as adding emergency retraction and status monitoring requirements in model V3, are highlighted through green borders around the objects and relationships within the model. Through identifying visualized model differences, model changes between versions are accurately captured and tracked, reducing the cognitive load on designers when updating complex models.

5.3. Scenario 2: Resolve design conflicts

When multiple designers are simultaneously working on different branches of a model, the tool-chain identifies and resolves design conflicts that exist in the different branch models. Figure 10 illustrates how the tool-chain identifies and resolves model conflicts during the merge process through a parameter merging scenario.

As shown in Figure 10(a), the system engineer combined BDD describe the system-level composition of the LGS and

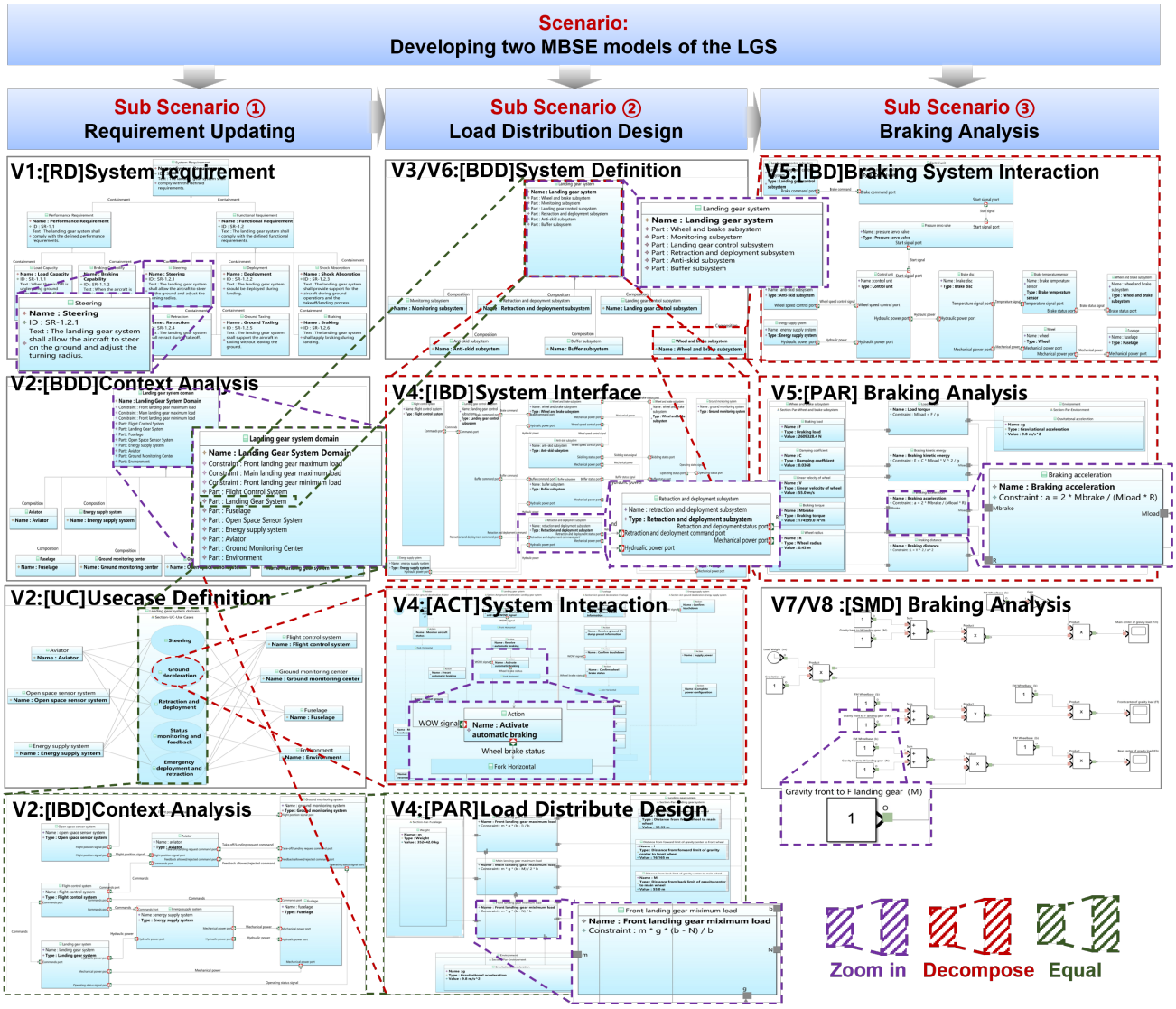

Figure 8: LGS models in MetaGraph

the definition of interface types in model V3. The LGS system includes the monitoring subsystem, the retraction and deployment subsystem, the landing gear control subsystem, the anti-kid subsystem, the buffer subsystem, and the wheel and brake subsystem. This model serves as the baseline for model V4 and model V5. In model V4, the SE further defines system-level interfaces, logical interactions between subsystems of LGS by the IBD and ACT, and then calculating load distribution of the LGS by the PAR. For example, the detail composition of the object *wheel and brake subsystem* includes brake actuator, wheel, pressure servo valve, control unit, brake temperature sensor, and brake disc that are shown in Figure 10(b). However, this composition refinement does not affect the current properties of the object *landing gear system*, but only the properties of the object *wheel and brake subsystem*. In model V5, through the braking analysis calculation defined in the PAR, and LE sets the design parameters (e.g., wheel radius and braking load) as value properties in the object *wheel and brake subsystem*. The braking torque is automatically calculated based on the input design parameters properties of the object *wheel and brake system* as shown in Figure 10(c). Thus, these aforementioned design parameters causes the property changes of both the object *landing gear system* and the object *wheel and brake subsystem*. Since we define the smallest unit in which conflicts can occur as an object in this paper, which implies that only models change differently in both branch in two branch simultaneously can potentially constitute a conflict. As a result, design conflicts arose between models V3, V4, and V5 regarding the properties of the object *wheel and brake subsystem*.

As illustrated in Figure 10(d), the SE and the LE each created a branch in the CDT platform and independently developed models on their respective branches. When the LE merged the developed Model V4 into Model V5 developed by the SE, the platform automatically detected whether there

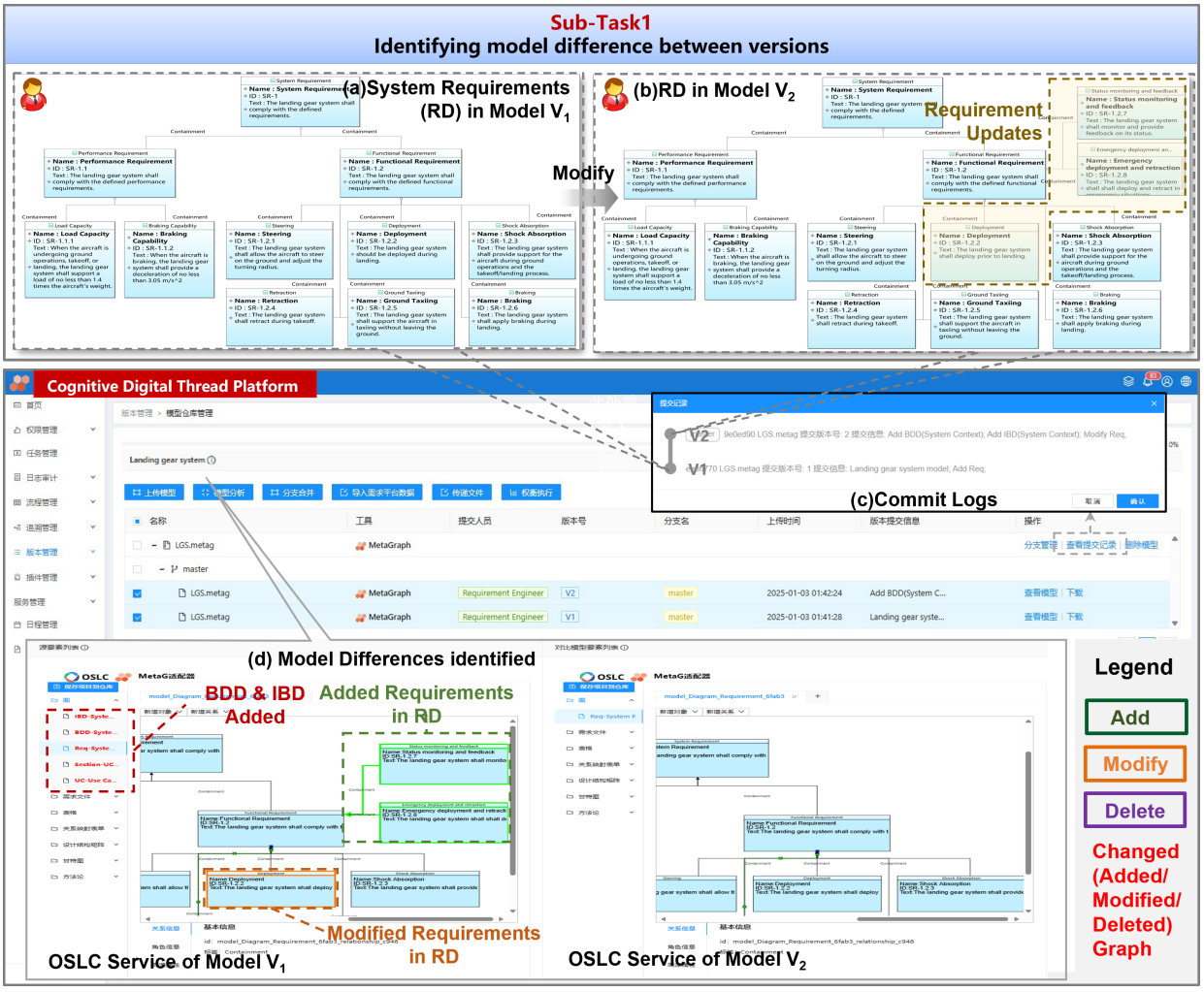

Figure 9: Scenario 1: Identify model difference

were design conflicts between the model and its common source model baseline. When design conflicts existed, it triggered the model merging process. The merging results are shown as three visualization UI windows provided by the OSLC service as shown in Figure 10(e). OSLC services of models V4 and V5 are located in the left and right windows, respectively, while the middle represents OSLC services of models V3 as their common source model baseline. Compared to the baseline model, both models V4 and V5 changed the properties of the object *wheel and brake subsystem*. Therefore, in the left and right windows, objects named *wheel and brake subsystem* are highlighted in orange to indicate changes. In the middle windows, the object *wheel and brake subsystem* is highlighted in red to indicate inconsistencies in the changes of these objects. Two possible operations for conflict solving are provided in the proposed tool-chain through OSLC services, which are synchronizing properties and modifying properties. When the LE clicks "Add to Merge Column" function button in the right-click menu of the object *wheel and brake subsystem* in model V4, its properties will be automatically reused and synchronized to the object *wheel and brake subsystem* in model V3. In addition, the LE can also click the property panel of the object *wheel and brake subsystem* in model V3, clicking the "Add property" button to flexibly configure the properties based on meta-models of BDD. For example, creating a *value property* for the the object *wheel and brake subsystem* as parameters such as the *wheel radius*, thereby resolving design conflicts of objects in the model at the property level.

5.4. Scenario 3: Analyze model versioning behaviors

After merging the aforementioned design conflicts, these merging and updating processes are synchronized and recorded in the knowledge graph of the tool-chain. Figure 11 illustrates how the tool-chain is used to analyze how models update and conflict between different versions.

In the modeling tool MetaGraph, the CDT platform provides a visual plugin to support the analysis of changes between different versions of models. Figure 11(a) shows the visual plugin in MetaGraph based on data in the knowledge graph stored in Neo4j. By selecting the modeling software

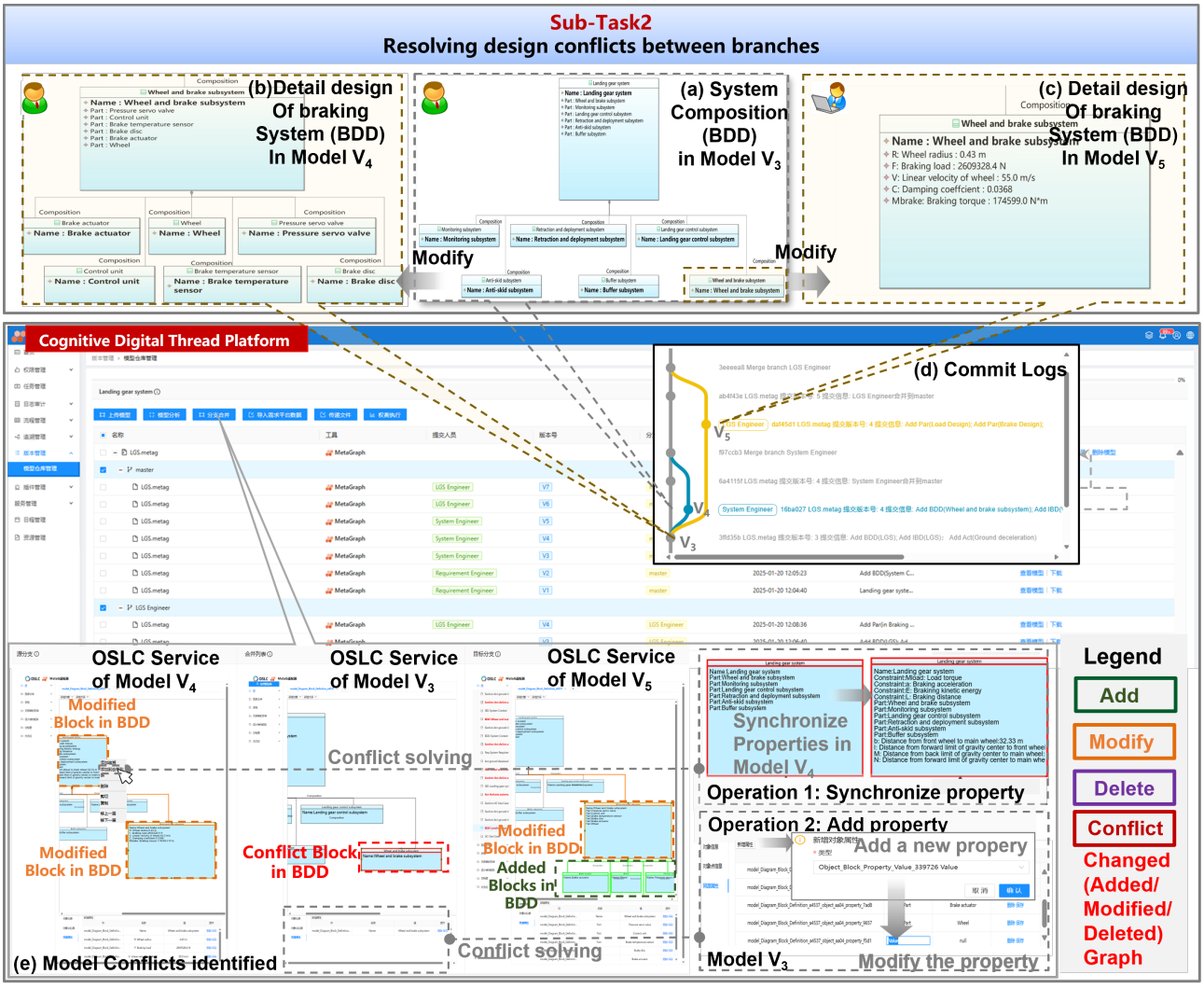

Figure 10: Scenario 2: Resolve design conflicts

tool used and the model name to be analyzed through drop-down menus, designers can get a model versioning diagram in the format of multiple hierarchical tree structures with relationships as Figure 11(b) shown. This diagram is generated by data stored in Neo4j database. Comparing to the flat representation of the knowledge graph in a node-relationship format, this model versioning diagram provides a structured analytical view and insight on the model evolution path for designer by combining and reusing all the query rules presented in Section 4.4. The information about hierarchical tree structure of the model is obtained by reusing matching rules 1, 2, and 3 in Section 4.4, while the version change relationship information between models is obtained by reusing matching rules 3 and 4 in Section 4.4.

In this plugin, each node in the hierarchical tree structure represents an OSLC service node in Neo4j. Figure 11(c) shows the three OSLC services corresponding to the design conflict mentioned in the scenario 2. By clicking on the node, it can be seen that the properties of the object *wheel and brake system* are modified from composition and parameters in model v4 and model v5, respectively. The object *wheel and brake subsystem* in different versions are connected by purple lines in the purple box of Figure 11(b), indicating that a design conflict has occurred between them. Figure 11(d) shows the two OSLC services corresponding to the change relationship between properties of constant block in SMD in model V7 and model V8. By clicking on the node, it can be seen that the property named *constant value* of the object *FM Wheelbase (b)* has been modified from 32 in model V7 to 32.33 in model V8, and these two objects are connected by a blue line, indicating that a modify relationship without conflicts has occurred between them. Through this plugin, designers can quickly understand the model evolution relationships between versions in the whole design process, obtaining detail model versioning actions between versions through OSLC services and knowledge graphs, further ensuring wise design decisions.

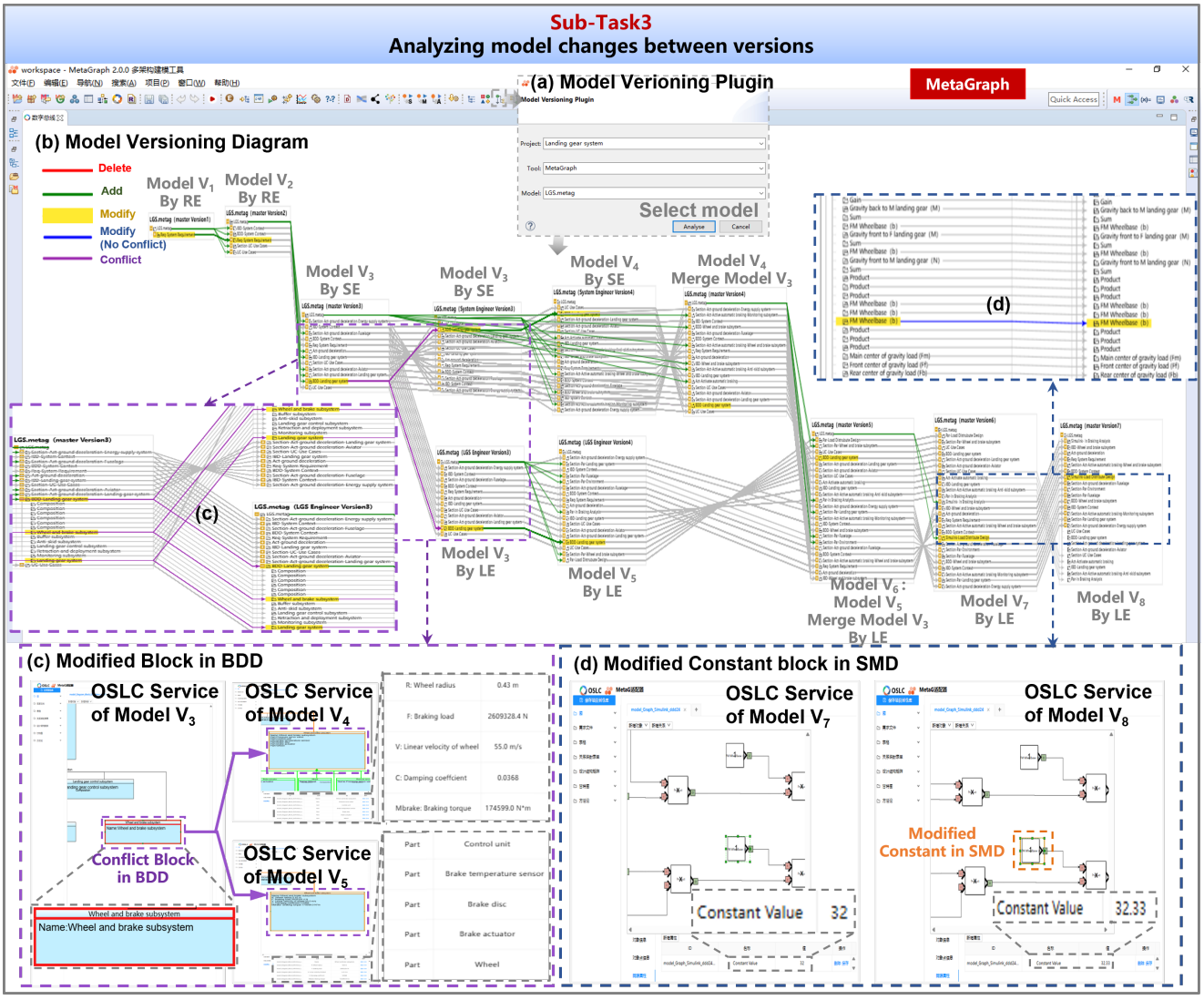

Figure 11: Scenario 3: Analyze model versioning behaviors

6. Discussion

In the MBSE domain, the CDT, as a semantic-enhanced DT, also requires maintaining a continuous and accurate representation of models throughout the entire system life-cycle. In the case study, we illustrate how the proposed tool-chain supports three key scenarios in model versioning. This tool-chain enables version management of models, ensuring all the changes and operations are tracked. The semantic information of MBSE models, such as system composition, behavior, and interfaces, is represented as OSLC services in the CDT through the OSLC adapter in the tool-chain, which is then further captured by the knowledge graph in the tool-chain. Through the relationships described in the knowledge graph schema and instances, the evolution of the semantic information of models can be precisely expressed, enhancing the DT's understanding of the impact of model version changes on other models. In this tool-chain, the collaborative participation of knowledge graphs, the modeling language KARMA, and OSLC adapters enables the model versioning of domain specific modeling languages, developing a CDT to enhance collaborative design capabilities of MBSE models in different modeling languages from the semantic perspectives.

In this section, we will further compare the model versioning process conducted by our tool-chain with that conducted by Git-based approach. We then evaluate both methods through quantitative experiments and qualitative criteria as mentioned in Section 3.2.

6.1. Quantitative evaluation

In this case study, we focus on **the increased efficiency due to the use of the proposed tool-chain in a model versioning process**. As introduced in Section 3.2, 4 metrics are used to assess the efficiency of model versioning. We compare measurements, interpret findings from results in Table 4 and draw conclusions.

Table 4
Measurement results of metrics of model versioning between two solutions

Approach	MC	MM(m)	LOC	T (s)
Our tool-chain	342	22.12	481	657
Git-based approach	405	34.96	847	1221
Reduced Operation	15.56%	36.73%	43.21%	46.19%

In Git-based approach, the MC and MM mainly come from the operations in local git repository (e.g., creating git repository folders) and operation in the MetaGraph (e.g., finding a conflicts and switching from git command line to MetaGraph to solve them). The LOC comes from the necessary command codes to input in git command line (e.g., git add, git commit, git push). In our tool-chain, the MC and MM mainly come from the file input and output operations and model-versioning-related operations in the CDT platform (e.g., uploading models, creating repositories, creating branches, solving conflicts, etc.). The LOC comes from the necessary version messages for each commit/branch/repository.

When we compare the MC and MM of these two approaches, we find these metrics with our tool-chain are 15.56%, 36.73% less than those with Git-based approach. We hypothesize the main reason is that in the Git-based approach, designers are not sure that their model versioning operations on model codes are accurately reflected in the model views, so they need to switch between the modeling tool and the versioning tool several times. In our tool-chains, we utilize OSLC services to provide graphical representation consistent with models in MetaGraph, and further support dynamic operations such as version update and conflict resolution based on OSLC services. This greatly enhances the interoperability and visualization ability of collaborative model versioning process. Thus, although our tool-chain may theoretically have more MC and MM during model versioning operations than a Git-based approach, the results demonstrate that our tool-chain reduces the operation of the designer to switching between modeling and versioning tools, and this decrease is obviously greater than the increase of operations caused by the increasing functions of the tool-chain itself.

When we compare the LOC of these two approaches, we find the LOC required for our tool-chain is 43.21% less than that required for the Git-based approach. On the one hand, in the Git-based approach, version update operations such as committing models need to be implemented by entering the corresponding git code (such as git commit, etc.). The designer needs to determine in real time which branch and which commit the current model is in, and then inputs corresponding git codes. In our tool-chain, these model versioning operations are encapsulated and shown as buttons, designers only need to check the current version status of the model according to the commit log diagram, and click the corresponding button to implement model versioning operations, which greatly simplifies the amount of LOC input. On the other hand, in the Git-based approach, configuration operations of model repositories and branches also need to be implemented by entering the corresponding git code (such as git checkout, etc.). In our tool-chain, the above operations are visualized as a graphical interface of model repositories and branches, and the designer can operate these repositories and branches through form-based configuration with key information , further simplifying the amount of LOC input required. As for the total time consumed, we find the metrics with our tool-chain are 46.19% less than that with Git-based approach. This can be explained by the fact that our tool-chain improves efficiency and reduces the time used during the whole model versioning process by providing a graphical representation consistent with the model in the modeling tool and simplifying the input code required in the model versioning process.

6.2. Qualitative Evaluation

Here we evaluate the proposed tool-chain using the four qualitative criteria mentioned in Section 3.2.

Domain-specific flexibility: As shown in the case study, the modeling languages adopted by MBSE models during model versioning usually incorporate features of different system characteristics, such as structures, parameters, and interfaces of LGS. In this paper, the GOPPRR-E modeling approach is used to develop MBSE models. This approach provides a better capability for configuring meta-models to describe system characteristics. When the tool-chain is adopted to conduct model versioning for models built by other modeling languages, the language adaptability of the proposed tool-chain can also be promoted, as it considers language-specific features such as syntax and conventions.

Capability of operation detection: When LGS developers change the MBSE models throughout the model versioning process, the nodes and edges in knowledge graphs increase. The increased nodes and edges characterize changes in MBSE model elements from the graph level to the property level, and these changes then provide insight into the model's evolution at different granularities. Compared with traditional state-based model comparison approaches, the knowledge graph's fine-grained operation detection ensures that we capture a user's model versioning behaviors at the model elements level. Such cognitive reasoning enables analysis of these behaviors, which thus helps maintain the fidelity and traceability between model elements along the versioning timeline.

Capability of Conflict Detection: When LGS developers change the MBSE models across versions and produce design conflicts, the three-way comparison algorithm together with OSLC services is used to identify the atomic conflicts in user operations. These OSLC services are developed in OSLC tool adapters for specific modeling tools, which means that more language-specific details of conflict detection can be captured. Besides, the developed OSLC services provide a graphical visualization for conflict detection in MBSE models, which helps developers easily understand the design conflict information.

Capability of Conflict Resolution: When LGS developers attempt to solve design conflicts, the employment of OSLC services provides them with the capability to manipulate the MBSE models, which reduces the effort and cost related to extending meta-models. This also promotes traceability between MBSE models and model versioning workflows. Besides, OSLC is a standardized specification for data integration among different design tools, which can promote the scalability of the proposed tool chain since we can integrate more design tools for other model versioning scenarios. Hence, we infer that our MBSE tool chain can improve the effectiveness of model versioning.

7. Conclusion

In this paper, we proposed a cognitive digital thread tool-chain for model versioning in model-based systems engineering. Through a case study of landing gear system design, the feasibility of the proposed tool chain is evaluated by both qualitatively and quantitatively. The results demonstrate that the proposed tool-chain has better efficiency than traditional Git-based model versioning approach. However, this study still has several limitations. First, the semantic completeness of the proposed tool-chain is not discussed in this paper. Formal verification of the functionalities of the tool-chain will be developed using ontology technologies in the future. Second, more complex model versioning behaviors, such as version rollback, are not discussed here. More model versioning patterns will be added in the future. Third, the tool-chain's external validity should be evaluated by more complex scenarios, such as those integrating system architecture models with CAD models.

8. Acknowledgements

This work was supported by the CN ministry project (Grant No. 50923010101) and the scholarship granted by the China Scholarship Council (No. 202406030154).

References

[1] Charles E. Dickerson and Dimitri Mavris. A brief history of models and model based systems engineering and the case for relational orientation. *IEEE Systems Journal*, 7(4):581–592, 2013.

[2] Junda Ma, Guoxin Wang, Jinzhi Lu, Hans Vangheluwe, Dimitris Kiritsis, and Yan Yan. Systematic literature review of mbse tool-chains. *Applied Sciences*, 12(7), 2022.

[3] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. *An Introduction to Model Versioning*, pages 336–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] Joeri Exelmans, Ciprian Teodorov, Robert Heinrich, Alexander Egyed, and Hans Vangheluwe. Collaborative live modelling by language-agnostic versioning. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 364–374, 2023.

[5] Edward Kraft. Hpcmp create™-av and the air force digital thread. In *53rd AIAA Aerospace Sciences Meeting*, pages 1–13, 2015.

[6] Foivos Psarommatis. A generic methodology and a digital twin for zero defect manufacturing (zdm) performance mapping towards design for zdm. *Journal of Manufacturing Systems*, 59:507–521, 2021.

[7] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[8] Shouxuan Wu, Jinzhi Lu, Zhenchao Hu, Pengfei Yang, Guoxin Wang, and Dimitris Kiritsis. Cognitive thread supports system of systems for complex system development. In *2021 16th International Conference of System of Systems Engineering (SoSE)*, pages 82–87. IEEE, 2021.

[9] Shouxuan Wu, Guoxin Wang, Jinzhi Lu, Zhenchao Hu, Yan Yan, and Dimitris Kiritsis. Design ontology for cognitive thread supporting traceability management in model-based systems engineering. *Journal of Industrial Information Integration*, page 100619, 2024.

[10] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, August 2009. Publisher: Emerald Group Publishing Limited.

[11] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, 13(1):239–272, 2014.

[12] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[13] Kaitlin Henderson and Alejandro Salado. Value and benefits of model-based systems engineering (mbse): Evidence from the literature. *Systems Engineering*, 24(1):51–66, 2021.

[14] Wang Zhe, Jerome Hugues, Jean-Charles Chaudemar, and Thierry LeSergent. An Integrated Approach to Model Based Engineering with SysML, AADL and FACE. pages 2018–01–1942, October 2018.

[15] Michela Munoz Fernandez. Using AADL to Enable MBSE for NASA Space Mission Operations. In *SpaceOps 2014 Conference*, Pasadena, CA, May 2014. American Institute of Aeronautics and Astronautics.

[16] Joeri Exelmans, Jakob Pietron, Alexander Raschke, Hans Vangheluwe, and Matthias Tichy. A new versioning approach for collaboration in blended modeling. *Journal of Computer Languages*, 76:101221, 2023.

[17] Benjamin Kruse and Mark Blackburn. Collaborating with openmbee as an authoritative source of truth environment. *Procedia Computer Science*, 153:277–284, 2019. 17th Annual Conference on Systems Engineering Research (CSER).

[18] Daniel Siegl. Bridging the gap between openmbee and git. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 465–466. IEEE, 2021.

[19] Hubert Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–84. Springer, 1998.

[20] David Wagner, So Young Kim-Castet, Alejandro Jimenez, Maged Elaasar, Nicolas Rouquette, and Steven Jenkins. Caesar model-based approach to harness design. In *2020 IEEE Aerospace Conference*, pages 1–13, 2020.

[21] Andrey Sadovykh, Wasif Afzal, Dragos Truscan, Pierluigi Pierini, Hugo Bruneliere, Alessandra Bagnato, Abel Gómez, Jordi Cabot, and

Orlando Avila-García. On a tool-supported model-based approach for building architectures and roadmaps: The megam@rt2 project experience. *Microprocessors and Microsystems*, 71:102848, 2019.

[22] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34, 2015.

[23] Steven Kelly. Collaborative modelling with version control. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 20–29. Springer, 2017.

[24] Don A Kinard. F-35 digital thread and advanced manufacturing. In *2018 Aviat. Technol. Integr. Oper. Conf*, pages 1–19. The F-35 Lightning II: From Concept to Cockpit, 2018.

[25] Mary A Bone, Mark R Blackburn, Donna H Rhodes, David N Cohen, and Jaime A Guerrero. Transforming systems engineering through digital engineering. *The Journal of Defense Modeling and Simulation*, 16(4):339–355, 2019.

[26] Timothy D West and Mark Blackburn. Is digital thread/digital twin affordable? a systemic assessment of the cost of dod's latest manhattan project. *Procedia computer science*, 114:47–56, 2017.

[27] Felipe F. De Oliveira, Icaro A. Fonseca, and Henrique M. Gaspar. Applying Open Web Architectures Towards Collaborative Maritime Design and Simulation. In *Volume 1: Offshore Technology*, page V001T01A013, Hamburg, Germany, June 2022. American Society of Mechanical Engineers.

[28] Stanislaus Reitenbach, Maximilian Vieweg, Richard Becker, Carsten Hollmann, Florian Wolters, Jens Schmeink, Tom Otten, and Martin Siggel. Collaborative Aircraft Engine Preliminary Design using a Virtual Engine Platform, Part A: Architecture and Methodology. In *AIAA Scitech 2020 Forum*, Orlando, FL, January 2020. American Institute of Aeronautics and Astronautics.

[29] Mary Bone, Mark Blackburn, Benjamin Kruse, John Dzielski, Thomas Hagedorn, and Ian Grosse. Toward an interoperability and integration framework to enable digital thread. *Systems*, 6(4):46, 2018.

[30] Jinzhi Lu, Guoxin Wang, and Martin Torngren. Design Ontology in a Case Study for Cosimulation in a Model-Based Systems Engineering Tool-Chain. *IEEE Systems Journal*, 14(1):1297–1308, March 2020.

[31] Thomas D. Hedberg, Manas Bajaj, and Jaime A. Camelio. Using Graphs to Link Data Across the Product Lifecycle for Enabling Smart Manufacturing Digital Threads. *Journal of Computing and Information Science in Engineering*, 20(1):011011, February 2020.

[32] Soonjo Kwon, Laetitia V Monnier, Raphael Barbau, and William Z Bernstein. Enriching standards-based digital thread by fusing as-designed and as-inspected data using knowledge graphs. *Advanced Engineering Informatics*, 46:101102, 2020.

[33] Moneer Helu, Alex Joseph, and Thomas Hedberg Jr. A standards-based approach for linking as-planned to as-fabricated product data. *CIRP Annals*, 67(1):487–490, 2018.

[34] Mouna Fradi, Faïda Mhenni, Raoudha Gaha, Abdelfattah Mlika, and Jean-Yves Choley. Conflict resolution in mechatronic collaborative design using category theory. *Applied Sciences*, 11(10), 2021.

[35] Donald J Treffinger, Scott G Isaksen, and K Brian Stead-Dorval. *Creative problem solving: An introduction*. Routledge, 2023.

[36] Jinzhi Lu, Junda Ma, Xiaochen Zheng, Guoxin Wang, Han Li, and Dimitris Kiritsis. Design Ontology Supporting Model-Based Systems Engineering Formalisms. *IEEE Systems Journal*, 16(4):5465–5476, December 2022.

[37] Dave Johnson and Steve Speicher. Open services for lifecycle collaboration-core specification version 2.0. http://open-services.net/, 2013.

[38] Helena Harrison, Melanie Birks, Richard Franklin, Jane Mills, et al. Case study research: Foundations and methodological orientations. In *Forum qualitative Sozialforschung/Forum: qualitative social research*, volume 18, 2017.

[39] N Salkind. Internal and external validity. *The SAGE dictionary of quantitative management research*, pages 148–149, 2011.

[40] Haoqi Wang, Vincent Thomson, and Chengtong Tang. Change propagation analysis for system modeling using semantic web technology. *Advanced Engineering Informatics*, 35:17–29, 2018.

[41] Jinzhi Lu, Jiqiang Wang, Dejiu Chen, Jian Wang, and Martin Torngren. A Service-Oriented Tool-Chain for Model-Based Systems Engineering of Aero-Engines. *IEEE Access*, 6:50443–50458, 2018.

[42] Heiko Kern, Axel Hummel, and Stefan Kühne. Towards a comparative analysis of meta-metamodels. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, page 7–12, New York, NY, USA, 2011. Association for Computing Machinery.

[43] Han Li, Guoxin Wang, Jinzhi Lu, and Dimitris Kiritsis. Cognitive twin construction for system of systems operation based on semantic integration and high-level architecture. *Integrated Computer-Aided Engineering*, 29(3):277–295, 2022.

[44] Rui Chen, Guoxin Wang, Shouxuan Wu, Jinzhi Lu, and Yan Yan. A service-oriented approach supporting model integration in model-based systems engineering. In *2023 IEEE International Systems Conference (SysCon)*, pages 1–7, 2023.

[45] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[46] Mingfei Liu, Xinyu Li, Jie Li, Yahui Liu, Bin Zhou, and Jinsong Bao. A knowledge graph-based data representation approach for iiot-enabled cognitive manufacturing. *Advanced Engineering Informatics*, 51:101515, 2022.