Version Management and Conflict Detection Across Heterogeneous Engineering Data Models

Florian Waltersdorfer  Thomas Moser  Alois Zoitl  Stefan Biffl
Christian Doppler Laboratory for Software Engineering Integration for Flexible Automation Systems
Vienna University of Technology, Austria
{florian.waltersdorfer, thomas.moser, alois.zoitl, stefan.biffl}@tuwien.ac.at

Abstract - Automation systems engineering projects depend on contributions from several engineering disciplines. These contributions consist of different artifacts, such as mechanical, electrical, and software components and plans, which get updated concurrently. While there are version management features in the software tools for each individual engineering discipline, there is very little work on version management across semantically heterogeneous data models in engineering tools and projects. In this paper, we introduce the Engineering Database (EDB) concept, which provides foundations for version management and update conflict detection in engineering data models across tool boundaries. We evaluate the concept based on a real-world use case for signal engineering at a hydro power plant systems integrator. Major result is that the parsing of proprietary engineering tool data exports can be generalized and the mappings between engineering tools can be simplified.

I. INTRODUCTION

Automation systems engineering projects depend on contributions from several engineering disciplines. While some engineering processes model contributions of engineers from different disciplines in a sequence of steps, in practice, the engineers concurrently update their artifacts, like mechanical, electrical, and software components and plans, to address new requirements and defect reports [6, 20].

Therefore, version management of engineering artifacts across tools and engineering disciplines is an important capability in mature multi-disciplinary engineering projects. Ideally, this kind of version management would (a) automatically identify conflicts between related data model elements when an engineer checks in his changed data model and (b) propose actions to resolve the versioning conflict.

In multi-disciplinary engineering projects for manufacturing systems, there exist common concepts, such as signals, on team level. However, the models in the engineering disciplines and their specific tools often use a range of terms and/or modeling structures to describe a given common concept, leading to semantically heterogeneous models [19]. Unfortunately, most engineering tools assume homogeneous data models, where concepts with similar meaning have also syntactically similar encoding. Therefore, these tools are hard to seamlessly integrate into an engineering environment on team level.

While there exist version management features in the software tools for each individual engineering discipline, there is very little work on version management across semantically heterogeneous data models in engineering tools. So-called “integrated” tool suites often consist of a predefined set of tools and a homogeneous common data model, which work well in their narrow scope but do not easily extend to other tools in the project outside the tools’ scope. However, system integrators in multi-disciplinary engineering projects want to be able to conduct change and version management across all tools that contribute project-level data elements regardless of the origin of the tool and data model.

In this paper, we introduce the Engineering Database (EDB) concept, which allows version management of data models across tool boundaries as long as the data elements can be linked to common domain concepts. The EDB concept maps the relevant data elements in local tool data models to the respective elements in a (virtual) common project/domain data model. These mappings allow deriving data transformers that translate updates from a specific tool data model to a new version of the common project data model. Therefore, it is possible (a) to check out the relevant changes of one tool into the data model of another tool and (b) to detect update conflicts of concurrent changes. In a change management process, where engineers check out a registered version from the project database, it is even possible to propose specific actions to resolve version conflicts.

We evaluate the concept based on a real-world use case for signal engineering at a hydro power plant systems integrator [20]. Signal engineering seems to be a good starting point in the broad scope of engineering model elements: signals are important synchronizing data elements at the interfaces between mechanical, electrical, and software engineering models. Therefore, mismatches in signals used in different engineering disciplines can easily lead to major risks for the project, if they go undetected.

For example, the correct modeling of signals links the information of a sensor in the field to the corresponding software variable, which is the foundation for describing the behavior of the system. If this link is not correctly modeled, defects may occur that are very hard to detect in any single engineering view. Version management of signals across tool boundaries is the foundation for strong quality assurance and risk management in a systems integration project.

The remainder of this paper is structured as follows: Section 2 summarizes related work on automation systems engineering, on version management, and on schema heterogeneity; Section 3 identifies the research issues. Section
4 describes the solution approach, while Section 5 presents the evaluation using a version management use case. Finally, Section 6 discusses the findings and identifies further work.

II. RELATED WORK

This section summarizes related work on automation system engineering, on version management, and on schema heterogeneity and solution approaches.

A. Automation Systems Engineering

Automation systems (AS), such as complex industrial automation plants for manufacturing, depend on distributed software to control the system behavior. The behavior of AS must be testable and predictable to meet safety and quality standards. In automation systems engineering (ASE) software engineering depends on specification data and plans from a wide range of other engineering aspects in the overall engineering process, e.g., physical plant design, mechanical and electrical engineering, and production process planning. This expert knowledge is embodied in domain-specific standards, terminologies, people, processes, methods, models, and software. The weak semantic integration of the expert knowledge across domain boundaries of engineering aspects makes changes late in the engineering process risky [4]. Thus the traditional ASE process follows a strongly sequential procedure and suffers from a lack of systematic feedback to earlier steps, low engineering process automation and weak quality management across domain boundaries, which leads to development delays and risks for system operation [14, 15].

In the heterogeneous ASE environments capabilities for effective and efficient integration of engineering systems and semantic integration of engineering knowledge are key enablers for flexible engineering process automation and advanced quality management. The engineering of complex AS typically involves several engineering domains, e.g., electrical engineering, mechanical engineering, and computer science, suffering from weak integration and semantic gaps between the engineering domains.

One of the most prominent problems in current industrial development and research approaches is the lack of model integration between the engineering disciplines [6, 19]. Different and partly overlapping terminologies are used in the different disciplines, which often hampers understanding. E.g., in software engineering, the word “version” usually refers to the change of a piece of software over time. In mechanical engineering, however, a “version” is a new product that is the result of a complex design and development process, such as the new version of a car [13].

B. Version Management

In software engineering, an important challenge in program development and maintenance is version control, i.e., the task of keeping a software system consisting of many components with different versions and configurations well organized [21]. Every software system exists in alternative forms, successive releases, or customer-tailored adaptations for different environments. The objective of a version control system is to maintain and control transformations made to a software system during its development and maintenance. Challenges of version control include controlling access to source files, storing, identifying, retrieving different versions of source and target files, and rebuilding target files when source files change [12].

In database research, schema versioning is defined as the general problem of using multiple heterogeneous schemata for various database-related tasks. In particular, schema versioning, and its weaker companion, schema evolution, deal with the need to retain current data and software system functionality in the face of changing database structures [18].

In software engineering research, version management is a key part of software configuration management. A big variety of version models has been realized in both commercial systems and research prototypes. These version models differ with respect to the objects put under version control (files, directories, entities, and objects), the organization of versions (version graphs versus multidimensional version spaces), the granularity of versioning, and the emphasis on states versus the emphasis on changes [22].

With the increasing employment of model-driven engineering techniques for software development, adequate infrastructural means supporting the effective management of software models become even more crucial and difficult. Tools successfully used for versioning textual artifacts like source code are, due to their line-oriented text comparison component, suitable for models only to a limited extent [1].

Although it is widely recognized that line-based Version Control Systems are not suitable for versioning model artifacts [2], this approach is often used in practical software projects and brings in several problems. In line-based approaches, the graph-based nature of models and therefore their syntax and even more problematic, their rich semantics are completely neglected. For example, if identical model elements are serialized in a different order, such line-based tools report changes wrongly. Moreover, XMI files are not intended to be read or interpreted by humans. However, line-based tools like Subversion are often used because of their general applicability. Nevertheless, such systems cannot prevent erroneous conflict detection, error prone conflict resolution, and inconsistent merged model versions. To tackle these shortcomings, versioning systems dedicated to the management of model artifacts are necessary [1].

C. Schema Heterogeneity and Solution Approaches

The problem of reconciling (data) schema heterogeneity has been a subject of research for decades, but solutions are few. The fundamental reason that makes semantic heterogeneity so hard to address is the independent origin of data sets using varying structures to represent the same (or overlapping) concepts [3, 8]. From a practical perspective, one of the reasons that schema heterogeneity is difficult and time consuming is that its solution requires both domain and technical expertise: a domain expert who understands the
domain meaning of all schemas being reconciled and technical experts for writing transformations.

Resolving schema heterogeneity is inherently a heuristic, human-assisted process. Unless there are very strong formal constraints on allowed schemas differences, one should not hope for a completely automated solution. Therefore, the goal is to reduce the time it takes human experts to create a mapping between a pair of schemas, and enable them to focus on the hardest and most ambiguous parts of the mapping [17].

There are several proposals for building semi-automated schema matching systems by employing a variety of heuristics [17]. The process of reconciling semantic heterogeneity typically involves two steps: In the first step, schema matching, and correspondences between pairs (or larger sets) of elements of the two schemas that refer to the same concepts or objects in the real world are identified. In the second step, these correspondences are exploited to create the actual schema mapping expressions.

Semantic integration is defined as the solving of problems originating from the intent to share data across disparate and semantically heterogeneous data [11]. These problems include the matching of ontologies or schemas, the detection of duplicate entries, the reconciliation of inconsistencies, and the modeling of complex relations in different data sources [16]. One of the most important and most actively studied problems in semantic integration is establishing semantic correspondences (also called mappings) between vocabularies of different data sources [9].

AutomationML (Automation Markup Language) is a neutral data format based on XML for the storage and exchange of plant engineering information, which is provided as an open standard [10]. A goal of AutomationML is to interconnect the heterogeneous tool landscape of modern engineering tools in their different disciplines, e.g., mechanical plant engineering, electrical design. AutomationML describes real plant components as objects encapsulating different aspects. An object can consist of sub-objects, and can itself be part of a bigger composition. An object can describe a screw, a claw, a robot, or a complete manufacturing cell in different levels of detail. The result of this analysis typically is a set of overlapping engineering concepts used in the engineering process. While this has proven to be true for well-established and long-running projects, the overlapping engineering concepts may not yet be known or elicited for projects in novel domains. In order to support future engineering projects, the knowledge regarding existing engineering projects should be used to derive guidelines to support similar future engineering projects.

III. RESEARCH ISSUES

In ASE projects several engineering disciplines contribute artifacts that get updated concurrently and contain data elements, which need to be synchronized with the content of corresponding data elements in data models of other tools. While a common understanding among engineers in a project on engineering domain concepts can be achieved, the tool data models are often semantically heterogeneous and therefore hard to synchronize automatically. For example, an electrical engineer creates a circuit plan for an oil pump controlling device which is concerned about in-and output current flows (only +5/-5V allowed). To the next higher level designer, who plans device wirings, this means that only binary/digital in-and output can be used on this machine. The third engineer involved, responsible for (fairly abstract) functional programming of hardware therefore has to know that only Boolean control parameters can be used in the corresponding section of the program. Since all three engineers are working with their own separated tools, data exchange is done informally. If either engineer fails to inform his/her colleagues about changes made to his design, the wiring/software will fail a later time for no obvious reason, which may lead to serious hardware damage or even human injuries. From this challenge we derive the following research issues.

R1a. Virtual common data model (VCDM). We investigate the design for a VCDM, which provides the foundations for version management of semantically heterogeneous tool data models effectively and efficiently.

R1b. Tool-specific transformers. The systematic derivation of transformers between the elements in tool-specific data models and corresponding VCDM elements.

R12b. Conflict detection. Approaches to conflict detection and resolution in engineering data models across tool boundaries based on version management.

We propose to adapt a combination of approaches from software engineering version management and semantic integration approaches to the needs of ASE projects with semantically heterogeneous engineering artifacts: the Engineering Database (EDB) concept. We evaluate the EDB concept based on a real-world use case for signal engineering at a hydro power plant systems integrator.

---

1 http://www.automationml.org/
The major precondition for using the EDB concept is a working communication link between the engineering tools to be integrated. An existing approach to (re-)integrate the tools has, among other solutions, led to the concept of the Enterprise Service Bus (ESB) [7]. Its idea is to provide a common infrastructure for tools to communicate with each other. However, current ESB implementations only provide limited possibilities for integration out of the box. The Open Engineering Service Bus (OpenEngSB)\(^2\) [5] aims at extending the capabilities of an ESB by introducing “tool domains”. Figure 1 gives an overview of the OpenEngSB’s architecture by displaying a set of engineering tools connected to the OpenEngSB, as well as some internal components of the OpenEngSB (such as the Engineering Database introduced in this paper, and the Engineering Workflow Rules), and additionally a set of runtime components, e.g., SCADA\(^3\) systems.

Any tool used is considered to have two properties: First, it is part of a workflow (else no integration would be required) and second, it can be replaced by another tool providing similar functionality (even if this may only be a newer or older version of the same tool). The first property implies that tool domains are in some relation to other tool domains which can be abstracted from the specific tools. The second property leads to the conclusion that every tool of a domain has a similar data model and provides certain services which are usually a single step of a workflow. Therefore tool domains can be used when modeling workflows with no need to explicitly address tools. This abstraction of course leads to the necessity of “tool connectors” used to connect a tool to its corresponding tool domain. With tool domains available, existing workflows can be improved, supervised and assisted more easily. This helps engineering groups to increase their efficiency by ensuring a correct and complete flow of information and a proper chain of actions. Creation and tracing of these workflows is required by engineering process engineers who describe workflows on a domain level (i.e., tool-independent) and by project and quality managers who need to trace and validate processes across tool boundaries.

IV. THE ENGINEERING DATABASE APPROACH

The Engineering Database (EDB) concept consists of (a) the data integration approach to align semantically heterogeneous tool-specific data models to a common domain model, (b) the version management approach based on data integration, and (c) the architecture and prototypical realization of tool support for data integration and version management.

For illustration of the approach we use signals in ASE, a basic design element that resembles specific I/O channels of automation devices. Every tool involved in the design process works with these signals, their tool-specific data formats, however, differ greatly.

D. Data integration approach

Figure 2 illustrates the structure of the EDB approach and the solution steps; the steps Ax correspond to the numbered square yellow tags shown in the figure.

A1. Derive the Virtual Common Data Model (VCDM).
The initial Virtual Common Data Model (VCDM) consists of all signals that are used in an interface between two engineering disciplines, e.g., from mechanical engineering to electrical engineering or from electrical engineering to software engineering. These signals can be elicited from the current engineering process as the signals can be found in lists that engineers exchange as part of the process. A facilitator can collect from the engineers the project-wide names for the concepts in the signals, for use in the VCDM.

In the VCDM for an example signal can be found at the bottom in the middle (white color).

Essentially, the VCDM is a project-wide repository of signals for the entire automation system, e.g., a power plant. An important characteristic of the VCDM is that it is built bottom-up from the information that the engineering tools share on project level. Therefore, the VCDM can be built incrementally and contains only the data elements necessary for version management.

A2. Mappings from tool-specific data models to the VCDM.
The second step is to map tool-specific data models (in yellow and green in Figure 2) to corresponding concepts the VCDM. This mapping contains (a) the logical links between tool-specific data elements and VCDM and (b) data format transformation instructions in both directions: tool-specific to VCDM and back.

From the logical mappings between the tool-specific data models and the VCDM it is straightforward to derive the code and configuration for the transformers that parse the data export from a tool for import into a project-wide repository in the VCDM format.

E. Version management and conflict detection

As shown in Figure 2 (steps Bx correspond to numbered green circles shown in the figure), using such transformation rules, the content is stored in the EDB matching the VCDM. For exporting data for a specific tool, an equivalent output parser needs to be defined.

B1. Data export. As a first step, data is exported from a proprietary tool data format to a neutral data format such as XML or CSV.

B2. Mapping to VCDM. Then, the exported data is mapped from the exported data in tool-specific semantics to semantics of the VCDM.

B3. Check in to EDB. A user updates data in her local tool and needs to check them into the EDB so her coworkers can work on the same version of the data.

\(^2\) http://www.openengsb.org
\(^3\) Supervisory Control and Data Acquisition
On check in, the check in-data is matched against the most actual state of the EDB by internally performing a checkout. Then, a difference analysis is performed; resulting in a listing of all signals, which do not yet exist in the EDB, and the signals, which have been updated. These matching checks are performed via a user-defined set of fields for identification, as not all tools allow storing custom data, such as the EDB's uuid⁴. This set of fields enables conflict resolution: the user manually selects which changes should be persisted into the EDB. Other users, who do not have this new data state, will receive this new state upon their next check in, which allows fixing conflicts from simultaneously edited fields of signals, avoiding update conflicts and simply updating non-conflicting data.

One of the most powerful features provided by using a Source Control Management (SCM) system like Git⁵ is the ability to go back in the history of the EDB until its creation and simply retrieve the "historic" data or reset the EDB to an earlier state. Unlike local tools or data bases, the reset/revert/undo/rollback operations of the EDB can be performed any time later, even after a complete server restart, as the EDB permanently stores the history of changes.

B4. Check out from EDB. On check out, a user may select sub-trees of the EDB, only receiving project-relevant data or even specific subsets of a project (e.g., for a specific hardware rack) and a tool-output parser. Now the user has a file for import into the given tool.

B5. Data import. Finally, the checked out data can be imported in the other local proprietary tool, using the data transformed from neutral data format to local tool data format in the previous step.

F. Architecture for tool support with the EDB

This subsection illustrates the tool components used for data integration in the EDB.

Parsers. The typical solution for data transformation in practice has been to use the tools initial export features (CSV files and/or XML-like structures) and apply manual and script-based conversions to match another tool's format.

---

⁴ uuid: universally unique identifier.
**EDB repository.** The Engineering Data Base (EDB) itself is a data storage based on Git\(^5\) and Apache Lucene\(^6\) for storing key-value-pair lists in a tree-based hierarchy, which provides high-performance indexing and versioning. Since the EDB is not based on a specific schema, the EDB allows inserting any kind of content, even if the underlying data model has been changed; see Listing 1 for an example entry.

```
uuid=3c1bdb0e-8c66-4b40-a7ee-fecdfb03ef1c
path=pathname/of/this/element
pathname=a
of=b
element=d
text=This is a text
text_alt=This is some more text
code=fe"HR/§&$/
```

**Listing 1:** Example EDB Entry.

The fields `uuid` and `path` are EDB-internal fields: `uuid` is a unique identifier, while `path` denotes the abstract pathname for an element. The fields named like the subparts of the abstract path will only be present if the actual path, in this example `a/b/c/d`, is given when inserting the entry into the EDB. Retrieving data is based on a subset of Lucene's query language, following the format presented in Listing 2.

```
[fieldname]:[partial or full keyname]
AND|OR
[fieldname]:[partial or full keyname]*)

```

**Listing 2:** EDB Query Syntax.

Listing 3 shows three EDB queries and their results.

```
uuid:3c1bdb0e-8c66-4b40-a7ee-fecdfb03ef1c
This will definitely return only this element
pathname:a AND of:b AND this:c AND element:d
This will return all elements under a/b/c/d
text:some more
This will return all elements with a field "text" containing at least "some more"
```

**Listing 3:** Example EDB Queries.

**Derive mappings from tools to VCDM.** After the VCDM is defined, a mapping to transform tool-specific data into this model and vice versa needs to be specified. Again, this step is based on existing knowledge, but different to typical point-to-point solutions, these mappings are based on the VCDM, which reduces the former m:n mappings (all tools to all other tools) to a 1:n mapping (all tools to the VCDM).

**Configure parsers with data mappings.** Once the mappings are sufficiently defined, new parsers are needed to actually transform the provided data (usually CSV-files) into a suitable format for persisting data into the EDB. To this purpose Smooks\(^7\) is used as a modern replacement for traditional scripting (e.g., VBA macros) and manual (e.g., Excel editing) parsing methods. The first example below describes one step of the parsing process: A field `functionTextOne` (textual description of a signal) from the CSV input (which has been pre-processed by the parser/transformation steps) is renamed to `FullText` in the VCDM. Listing 4 shows this example in XML syntax.

```
<!-- Create an "propBean" bean instance of the type hash map when we visit the start of the <csv-record> element which is returned by the csv-parser. This bean is wired into the "signalBean" ContentEntity. -->
<jb:bindings beanId="propBean"
class="java.util.HashMap" createOnElement="csv-record">
  <jb:value property="FullText" data="csv-record/functionTextOne" />]
</jb:bindings>

```

**Listing 4:** Parser Data Mapping Example - renaming a value.

The second example shows a more complex parser step, usually done right after basic textual extraction from the tools’ export dump: A text element named `identMixedValue` (a tool’s internal naming for a pair of hardware address parts) is extracted and split into two nodes `pBoardAddress` and `IO-Module`, which are the known standardized names of these fields. Listing 5 shows this example in XML syntax.

```
<!-- The parser part in this file uses the Groovy capabilities of Smooks to do some intelligent parsing of the input files. Groovy is a Turing complete scripting language and allows mighty operations on different nodes. They can be extended as needed and be used for table lookups and every other kind of operation. -->
<g:groovy executeOnElement="identMixedValue">  
  <g:script>  
    Document doc = elem.getOwnerDocument();  
    Element first = doc.createElement("pBoardAddress");  
    Element second = doc.createElement("IO-Module");  
    String all = elem.getTextContent();  
    first.setTextContent(all[0]);  
    second.setTextContent(all[1]);  
  </g:script>
</g:groovy>

```

**Listing 5:** Example EDB Queries.

---

\(^{5}\) http://git-scm.com/  
\(^{6}\) http://lucene.apache.org/  
\(^{7}\) http://www.smooks.org/
In order to evaluate potential improvements of the proposed VCDM and the Engineering Data Base (EDB) approach, we compared the proposed approach with traditional processes used at our industry partners. The two major advantages are on the one hand side the application of a single software tool for the parsing of the data export structures of the proprietary engineering tools, and one the other hand side the reduction of the needed mappings. In the following, we briefly assess the impact of these advantages. In addition, we revisit the initial scenario from section III to illustrate the changes to the “old” workflow.

Parsing of proprietary engineering tool data exports.

The traditional approach for data transformation was to use the engineering tools initial export features (CSV files and/or XML-like structures) and then to apply a mixture of manual and script-based conversions, such as VBA macros or Excel editing, to match another tool’s format. In the proposed approach, the Smooks parser is used as a generic replacement for former scripting and manual parsing methods.

Mappings between engineering tool data.

In the traditional approach, the engineers needed to create and maintain mappings between mutually all involved engineering tool data structures. In the proposed approach, there is only a single mapping required between each engineering tool and the VCDM, effectively reducing the former m:n mappings (all tools to all other tools) to a 1:n mapping (all tools to the VCDM), reducing the data integration effort from exponential to linear growth.

User experience reporting higher efficiency.

Feedback from a selected test group of power users at our industry partners showed that tool data exchange via the EDB takes a few minutes, including error corrections. The same experts stated that setup and execution of previous tool data exchanges had been scheduled to range from hours up to days. To verify these improvements, we examine the new workflow: The electrical engineer commits his/her work to the EDB where it is translated into the virtual common data model (VCDM). As soon as the device designer performs a checkout (which happens automatically before he can check-in his data), he is notified about the changes applied to the signals, realizing any “dangerous” changes, as those may cause inconsistencies in combination with his local tool. Therefore the engineer will fix or report them and commit his adaptations afterwards. Other roles also are able to see these changes and can react likewise. Since this is easier then with the previous semi-automatic approach, it is done more often and therefore on smaller data sets as before.

VI. DISCUSSION AND CONCLUSION

Automation systems engineering projects depend on contributions from several engineering disciplines. While there are version management features in the software tools for each individual engineering discipline, there is very little work on version management across semantically heterogeneous data models in engineering tools.

In this paper, we introduced the Engineering Database (EDB) concept, which provides the foundations for version management and update conflict detection in engineering data models across tool boundaries.

We derived the following research issues, which we addressed in a real-world use case for signal engineering at a hydro power plant systems integrator. Major result is that the parsing of proprietary engineering tool data exports could be generalized and the number of required mappings between engineering tool data was reduced.

R11a. Virtual common data model. Essentially, the VCDM is a project-wide repository of signals for the entire automation system, e.g., a power plant. An important characteristic of the VCDM is that it is built bottom-up from the information the engineering tools share on project level. Therefore, the VCDM can be built incrementally and contains only the data elements necessary for version management.

R11b. Tool-specific transformers. In the proposed approach, the Smooks parser is used as a generic replacement for scripting and manual parsing methods. Additionally, there is only a single mapping required between each engineering tool and the VCDM, effectively reducing the former m:n mappings to 1:n mappings.

R12b. Conflict detection. Conflict detection is based on a difference analysis which is performed on check in of new data into the EDB. This enables conflict resolution by highlighting conflicts and allowing the user to manually select which changes should be persisted into the EDB. Other users will receive the new data state upon their next check in, thus being able to fix any conflicts within simultaneously edited fields of signals, avoiding update conflicts and simply updating non-conflicting data.

Future work. Our initial study showed that the EDB concept works well and the engineers found the pilot prototype useful. Future research will include an explicit specification of the used engineering tool data structures, the so-called engineering concepts, using semantic modeling in order to support the automation of engineering process steps.
ACKNOWLEDGMENT

This work has been supported by the Christian Doppler Forschungsgesellschaft and the BMWFJ, Austria. This work has been partially funded by the Vienna University of Technology, in the Complex Systems Design and Engineering Lab. The authors also want to thank the domain experts at their industry partners for their support with the use case data, discussions, and feedback to the proposed approach.

REFERENCES