Introducing Traceability and Consistency Checking for Change Impact Analysis across Engineering Tools in an Automation Solution Company: An Experience Report

Andreas Demuth*, Roland Kretschmer*, Alexander Egyed*, Davy Maes†
*Institute for Software Systems Engineering (ISSE)
Johannes Kepler University (JKU)
Linz, Austria
{andreas.demuth, roland.kretschmer, alexander.egyed}@jku.at
†Flanders Make
Belgium
{davy.maes}@flandersmake.be

***Abstract*—In today's engineering projects, companies continuously have to adapt their systems to changing customer or market requirements. This requires a flexible, iterative development process in which different parts of the system under construction are built and updated concurrently. However, concurrent engineering is quite problematic in domains where different engineering domains and different engineering tools come together. In this paper, we discuss experiences with Van Hoecke Automation, a leading company in the areas of production automation and product processing, in maintaining the consistency between electrical models and the corresponding software controller when both are subject to continuous change. The paper discusses how we let engineers describe the relationships between electrical model and software code in form of links and consistency rules; and how through continuous consistency checking our approach then notified those engineers of the erroneous impact of changes in either electrical model or code.**

***Keywords*—Software product lines, safe composition, incremental consistency checking, model evolution**

I. Introduction

Today, technical development projects commonly span across multiple domains and disciplines. It has become common that engineers with diverse backgrounds collaborate in order to develop a complex system. For example, developing a cyber-physical system typically requires software engineers, hardware engineers, electrical engineers, requirements engineers, mathematicians, physicists, and others to work together closely. However, typically experts from these different domains are responsible for different development artifacts. For instance, the requirements the system under development must fulfill may be negotiated with stakeholders by the requirements engineer. The requirements may then be used by a mathematician to calculate data that must be known to realize the project. This data may then, in turn, be used by an electrical engineer to create a model of the hardware setup. Based on this hardware setup and other calculated data, a software engineer may, finally, develop the source code that controls the hardware components. Indeed, during initial development of the mentioned artifacts there is typically communication between experts from different domains. For example, once the hardware design is finished, the hardware engineer informs the software engineer about which hardware components will be used and thus have to be controlled by the control software. Each engineer will also quite likely inform the corresponding supervisor about a finished artifact. Therefore, during initial development, knowledge is typically passed on quite well between involved engineers and the different artifacts are likely to be consistent with respect to each other. However, such communication often happens informally, and knowledge passed during this communication might not be documented. This becomes a problem especially when an artifact is evolved past its initial stage, for example in an iterative development process in which individual artifacts are evolved multiple times until they reach their final state when the project is finished. In this case, the rationale for certain decisions is often no longer available for engineers. For instance, it might no longer be clear why certain source code fragments are needed for implementing the controlling of a piece of hardware. If an engineer removes a component from the hardware model, software engineers might have a hard time locating the corresponding source code fragments. As a consequence, changes because of evolution or maintenance may easily introduce inconsistencies between different artifacts and the impact of a change is hard to estimate.

These are exactly the issues that the Belgian company Van Hoecke Automation (VHA), a leading provider of automation solutions, is facing. In this paper, we present the results of a collaboration between VHA, Flanders Make, and JKU's ISSE that addresses these issues. In particular, we describe a solution that enables traceability, consistency checking, and change impact analysis between different artifacts in VHA's development process. The solution relies on existing principles of traceability and incremental consistency checking (e.g., the *DesignSpace Information Integration Platform* [1] and the *Model/Analyzer Consistency Checking Framework* [2], [3]). Moreover, new technologies for establishing traceability and

the application of consistency checking to different artifacts have been developed.

II. Background

Van Hoecke Automation is a leading Belgian Family company, founded in 1990, providing advanced solutions for production automation, product processing, and product inspection and tracking. Their main sectors of operation include the make industry with an important international activity in the automotive and OEM sector, and the food industry. Key competences of VHA regarding the discipline of software engineering include the development of production supervisory systems by means of standard tools and the development of customer specific machine applications. As it is typical for companies dealing with cyber-physical systems and production automation, VHA employs experts from diverse domains, such as business and management, software engineering, mechanical engineering, electrical engineering, math, or physics.

Flanders Make is the strategic research centre for the manufacturing industry with establishments in Lommel and Leuven. The centre collaborates with research labs at various Flemish universities as well as with major companies and SMEs. Together, they focus on product and process innovation based on the challenges and needs of the industry. The research focus is on 4 technological domains: power electronics and energy storage, mechatronics and design methods, production processes, and people-driven system development. The primary goal of the collaboration is to yield product and process innovations in 3 fields of application: vehicles, machines, and factories.

In this project, VHA and Flanders Make collaborated with JKU's ISSE to combine their experience in the development of mechatronical systems with leading research on software traceability and consistency checking. The results are meant to not only be beneficial for the industrial partners with respect to their development process, but also to serve as case study for the application of research technologies from academia in an industrial context.

III. Problem Statement

For this project, VHA and Flanders Make provided a sample scenario illustrating an issue that occured during recent iterative development. Next, we describe in detail this scenario and the issue associated with it.

The scenario is about the design of a conveyor belt which is powered by an asynchronous motor. The motor is controlled by a programmable logic controller (PLC). This scenario involves four different engineers: i) a systems engineer, ii) an electrical engineer, iii) a safety engineer, and iv) a software engineer. The process is depicted in Fig. 1. Each engineer is using specific tooling during the following individual steps of the scenario:
1) [System concept] The systems engineer creates the specification of the initial system concept (as shown in Fig. 2(a)).
2) [Electrical design] Eplan Electric P8 is used by the electrical engineer to specify the electrical connections between actuators (e.g., a motor) and the PLC (a high-level view of the electrical design is shown in Fig. 2(b), a snippet from the actual EPlan P8 model is shown in Fig. 2(c)). The EPlan P8 model snippet depicts the control signal of the conveyor motor

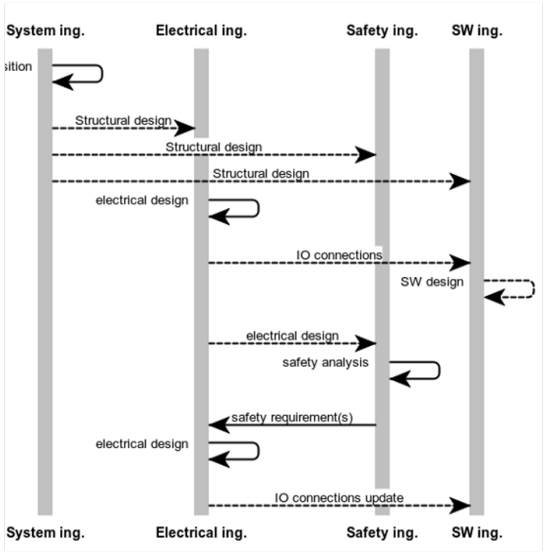


Fig. 1: Incremental development of conveyor belt system.

as the component `21K3` and the normal output of the Siemens PLC as the component `Q70.0`.
3) [Software design] In the example, the Eclipse IDE with Java is used as a proxy for the regular PLC code in which the control logic of the actuators are implemented (as shown in Fig. 2(d)). Notice that for the two IO ports of the Siemens PLC, there are two variables defined: i) `hw_IO_out0_1` representing the normal output (`Q70.0` in Fig. 2(c)), and ii) `hwIO_out1_1` representing the safety output of the Siemens PLC (not depicted in Fig. 2(c)). Moreover, the variable `fb_motor_control` represents the control signal of the motor conveyor (`21K3` in Fig. 2(c)). In the last line of Fig. 2(d), the linking of the components `Q70.0` and `21K3` from Fig. 2(c) is reflected.
4) [Safety analysis] Microsoft Excel is used by the safety engineer for the analysis of potential harms to (human) operators caused while operating the system, using a template specifically designed at VHA (as shown in Fig. 2(e)).
5) [Electrical design update] the electrical engineer performs updates of the electrical connections to avoid the harms identified in the previous step (as shown in the high-level view in Fig. 2(f) and the actual EPlan P8 model snippet in Fig. 2(g)). Notice that in Fig. 2(g) the control signal (component `21K3`) is now connected to a different component (`Q30.1`), which represents the safety output of the Siemens PLC.
6) [Software design update] the software engineer performs updates of the software so that it is in sync with the updated electrical connections (as shown in Fig. 2(h)). Notice that in the mapping of IO ports, the control signal (`fb_motor_control`) is now linked to the safety output of the Siemens PLC (`hwIO_out1_1`).

Note that after Step 5, the update of the electrical design, there is an inconsistency between the electrical design and the control software, and thus an update of the software design is required. Specifically, in the EPlan P8 model snippet in Fig. 2(g) it can be seen that the component `21K3` is no longer connected to the component `Q70.0` (as in Fig. 2(c)), but instead it is now connected to the safety output of the

Siemens PLC, which is modeled as the component `Q30.1`. The source code from Fig. 2(d) still reflects that `21K3` and `Q70.0` are connected. However, VHA has experienced that in such a scenario the inconsistency may be overlooked by engineers and may thus remain undetected, resulting in Step 6 being not performed in practice. This might be caused by a lack of support for traceability, change impact analysis, and inconsistency detection. The consequences of not detecting the inconsistency and not performing step 6 in the scenario include, for example, delays in the development progress or failure to deliver a functioning system. Therefore, it is crucial for VHA to introduce support for traceability, change impact analysis, and inconsistency detection in their development process.

IV. Goals and Requirements

Let us now summarize the goals and specific solution requirements stated by VHA. The main goals of the project presented in this paper are:

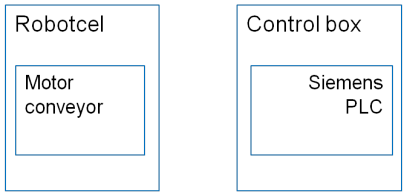
- Increase awareness about consistency between spreadsheet data, source code, and electrical models,
- Increase traceability between these development artifacts, and
- Get feedback about consistency and change impact.

A key requirement stated by VHA is the support for the following technologies: Microsoft Excel Spreadsheets, and Eplan Electric P8. Additionally, in the example, the Eclipse IDE with Java is used as a proxy for the regular PLC code development. Engineers using either tool must be notified about arising inconsistencies. Moreover, it must be possible to analyse the impact of a change and to track dependencies between different artifacts. Additionally, the current workflows of the individual disciplines should not be disrupted significantly (i.e., the specialized engineering tools used currently must not be exchanged). Now that we have illustrated VHA's challenge problem and discussed the key goals and requirements they stated, we will next present the solution that was developed in the project.

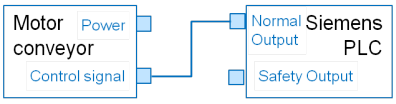
V. Realization

In order to address the challenge problem presented above, we decided to develop a solution that is based on existing approaches and that does not interfere with VHA's established development process. In this section, we discuss the general architecture of our solution and then describe in detail how the individiual parts work.

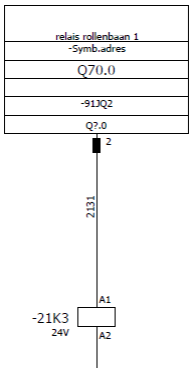
As discussed above, incremental consistency checking and traceability between different artifacts are key features our solution must provide. The original development artifacts (e.g., Excel files) should remain unchanged. Therefore, they are duplicated in the employed *DesignSpace data integration platform* [1]. Two services running on top of the DesignSpace platform are then used for performing consistency checking and enabling traceability between different existing artifacts. Next, we first discuss how artifact integration works in our solution, then present how traceability can be established using the integrated data, and finally show how consistency checking



(a) Initial system concept from Step 1.



(b) Initial electrical design from Step 2.



(c) Initial EPlan P8 model from Step 2.

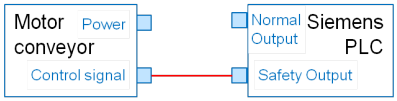
```
//Q70.0
int hwIO_out0_1 = 0xc001;
//Q30.1
int hwIO_out1_1 = 0xc001;
//21K3
int fb_motor_control = 0xc002;

//linking
IO_mapping.put(hwIO_out0_1, fb_motor_control);
```

(d) Initial software design from Step 3.

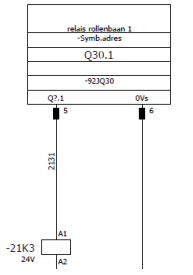
Danger	Likelihood	frequency	...	Corrections
Pinched in between conveyor belt	Probably	Daily	...	fail safe on conveyor motor

	Required SRP/CS level			
Safety Functions	S	F	P	SILr
Fail safe conveyor motor	S2	F1	P2	2

(e) Initial safety analysis from Step 4.



(f) Updated electrical design from Step 5.



(g) Updated EPlan P8 model from Step 5.

```
//Q70.0
int hwIO_out0_1 = 0xc001;
//Q30.1
int hwIO_out1_1 = 0xc001;
//21K3
int fb_motor_control = 0xc002;

//linking
IO_mapping.put(hwIO_out1_1, fb_motor_control);
```

(h) Updated software design from Step 6.

Fig. 2: Development artifacts of conveyor belt system. Initial versions (a)–(e) and updated versions (f)–(h).

is realized using the integrated data and existing traceability information.

A. Artifact Integration

As discussed above, the DesignSpace integration platform is used in this project. This platform provides support for the transparent integration of arbitrary development artifacts. It also supports additional services for establishing traceability between arbitrary development artifacts and incremental consistency checking. The DesignSpace internally uses a single concept of representing information as attributed graphs and it does not require development artifacts to be moved to a single point of storage. Instead, all development artifacts remain unchanged at their original location (e.g., on an engineers local harddrive) and the DesignSpace only holds a replication of the original information that is synchronized live. Thus, artifact integration with the DesignSpace does not affect existing processes and practices. Engineers are still able to use their favorite engineering tools for developing their artifacts. This is important for two reasons. First, engineers are typically reluctant to changing the used tools, which is understandable because engineering tools are typically highly specialized for performing certain tasks as efficient as possible. Thus, trying to replace such highly specialized tools is typically a non-optimal solution with respect to the quality of results and the efficiency these results are produced. Second, developing a new engineering tool for a specific domain requires years of development, with only small chances of coming up with a tool that leads to better—or even similar—results than the leading existing tools for the domain. Notice that any information is stored using the DesignSpace's internal data structure (i.e., a graph-like structure).

Integrating data with the DesignSpace platform can be achieved in different ways. One option is to use file-adapters that translate file contents into the DesignSpace's internal data structure. The second option is to use tool-adapters that directly translate and synchronize information from engineering tools with the DesignSpace. Note that the latter can be done live and incrementally while engineers work on their respective development artifacts, whereas with file-based integration the synchronization of the original development artifact and its duplicate in the DesignSpace can only be done when the file is saved, requiring the use of diffing tools or inefficient overrides of existing data. Moreover, by using tool-adapters it is possible to present information about consistency or trace information to the engineer directly in the engineering tool. Therefore, the use of tool-adapters is generally preferable.

For each of the three tools that are used by VHA in their challenge problem, we developed three different tool adapters, which we present next.

1) Microsoft Excel: For Microsoft Excel, data integration is performed at cell level through an Excel plugin written in C#. We decided to not duplicate all cells in a spreadsheet with the DesignSpace, but only those cells that are of relevance for other development artifacts. This decision is based on the fact that, due to the nature of spreadsheets, a vast majority of existing cells may be either empty or contain information that is required only for some internal calculations but does not have any effect on other artifacts. Instead, those cells in the spreadsheet that are of relevance and should therefore be synchronized with the DesignSpace for later use by the traceability and consistency checking services are annotated by an engineer. This avoids unnecessary communication overhead and allows for easier establishing of traceability, as we will see below. The annotations are added to each cell individually using Excel's comment function. To distinguish comments for data integration from other comments, data integration comments must conform to the following syntax: `$name=<name>; $unit=<unit>; $meta=<meta information>`. The `$name` attribute allows engineers to specify a name by which the cell can later be references for traceability and consistency checking. The `$unit` attribute defines the unit of the value saved in the annoted cell. The attribute `$meta` allows engineers to specify additional meta information, for example if a cell is representing some domain concepts. As an example, consider in Fig. 2(e) the cell with the value `Probably` (2nd row, 2nd column). Using the `$meta` attribute, an engineer can define that this cell's value represents a likelihood. Note that another option would have been to use predefined templates that define which cells should be synchronized. However, using annotations provides more flexibility to engineers in case templates are changed.

2) Eclipse IDE with Java: For source code development, in this project we relied on the Eclipse IDE. Therefore, we developed a tool-adapter plugin for the Eclipse IDE and the Java programming language. Notice again that in practice PLCs are not programmed using Java. However, in this project we agreed with VHA to use Java for demonstrating the feasibility of checking consistency between EPlan P8 models and source code. VHA is confident that if consistency checking is feasible with Java, it is also feasible with an actual PLC programming language. Similarly to the tool-adapter for Microsoft Excel, this adapter does not synchronize the entire source code with the DesignSpace platform, but only those parts of it that are relevant for other artifacts. As for excel, this reduces the amount of information that is available in the DesignSpace for establishing traceability. Engineers can mark relevant parts by adding annotations as comments. Note that such annotations can be made for any element of a program (e.g., classes, methods and functions, individual statements, etc.). The synchronization with the DesignSpace is not performed immediately when an annotation is added, but it is performed each time the Eclipse IDE's compiler is executed, which is typically after each save triggered by the user.

3) EPlan Electric P8: For electrical models created with EPlan Electric P8 we followed a similar approach as for Microsoft Excel and the Eclipse IDE with Java. A tool-adapter is used that observes the electrical model. Any model elements that should be synchronized with the DesignSpace have to be marked by adding a custom property to the element (e.g., to a modeled component). However, in EPlan it is not possible to select connections between elements and set this property. Therefore, the tool-adapter also synchronizes all connection elements that connect two (or more) already marked (and therefore synchronized) elements. For instance, if a motor element is connected to a controller element, then the connection between the motor and the controller is also synchronized with the DesignSpace. An example of this behavior is shown in Fig. 3. Specifically, the left-hand side of Fig. **??** shows three model elements that are synchronized with the DesignSpace:

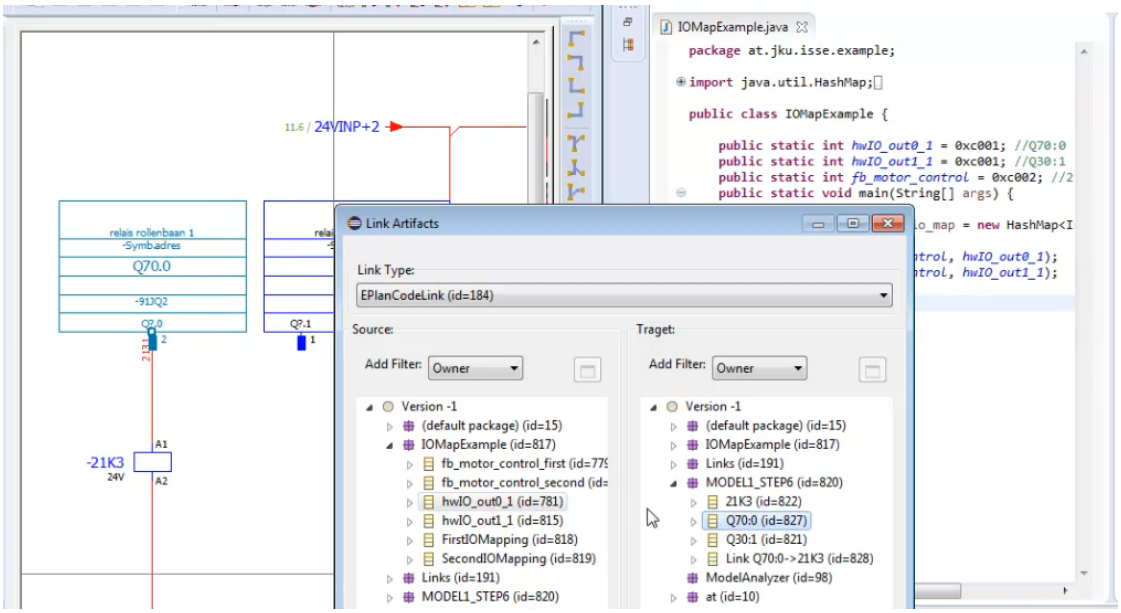


Fig. 3: Wizard for creating a link between EPlan model element and its corresponding source code fragment.

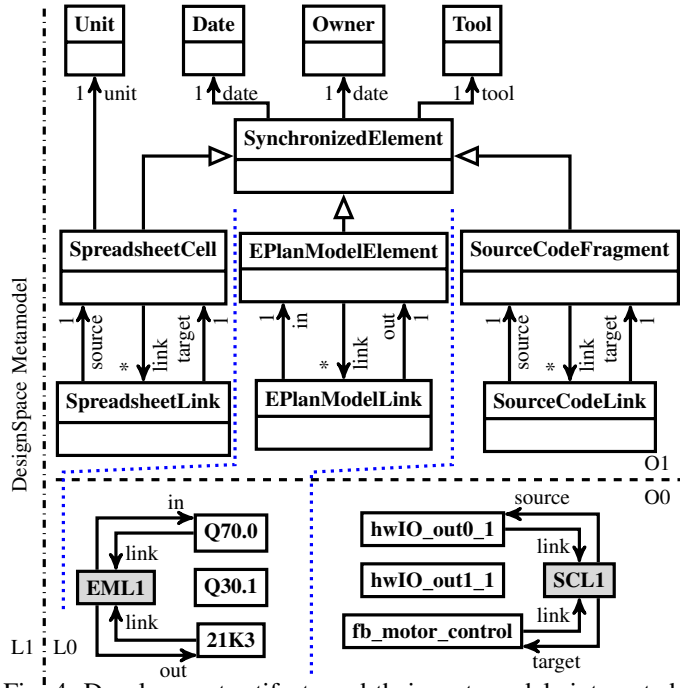


Fig. 4: Development artifacts and their metamodels integrated in the DesignSpace.

`Q70.0`, `Q30.1` (only visible partially to the right of `Q70.0`), and `21K3`. The corresponding DesignSpace representation of the EPlan model is shown on the right-hand side of the linking wizard in the center of Fig. 3. Notice that there are four entries: one for each of the marked model elements, and one additional representation of the link between `Q70.0` and `21K3`. This link has been created in the DesignSpace automatically.

4) Integrated Metamodels of Development Artifacts: As discussed above, the DesignSpace platform internally uses a simple, graph-like data structure that represents any information as artifacts and properties. To still have available type information for the development artifacts represented in the DesignSpace, not only the actual development artifacts (e.g., EPlan models or source code) are synchronized with the DesignSpace, but also their metamodels. While the development artifacts are modeled as logical instantiations of their metamodels, the representations of both, models and their metamodels, are runtime instances of the DesignSpace's metamodel. Therefore, it must be distinguished between linguistic and ontological levels. Linguistic levels define technical instantiation at runtime, whereas ontological levels describe logical instantiation.

In Fig. 4, the linguistic and ontological levels are depicted. Notice that we use the same notation for representing the individual development artifacts at the ontological level `O0` and their respective metamodels at the ontological level `O1`. This indicates that both levels are runtime instances of the linguistic level `L1`. The distinction between the two ontological levels is purely virtual and based on interpretation of the runtime data structures. For each integrated development artifact its corresponding metamodel remains unchanged in principle—the metamodel may only be extended with additional information, but the original concepts are retained. This means that any issues are avoided that typically would arise when trying to merge different metamodels into a single one [4], [5]. However, the metamodels used to provide typing information of synchronized development artifacts has been simplified for this project. This was not only done to simplify the development of tool adapters, but also to reduce the complexity of linking and consistency checking from the perspective of the involved engineers. Specifically, the metamodel of EPlan models contains only two types: `EPlanModelElement` and `EPlanModelLink`. The for-

mer is used to represent any first-class entities of EPLan models (e.g., components), whereas the latter is used for representing any links that are drawn between first-class entities (e.g., when an input of a component is linked to another component's output). For source code, the metamodel contains also two classes: `SourceCodeFragment` and `SourceCodeLink`. A `SourceCodeFragment` can be any part of source code, for example a class, a method, an individual statement, or even a simple comment. A `SourceCodeLink` is used to model linking and mapping of source code fragments. For instance, a `SourceCodeLink` between two variables is used to model that the two variables are used as key-value pair in a map. For Microsoft Excel spreadsheets, the tool adapter only synchronizes individual cells. Therefore, a single metamodel element, `SpreadsheetCell` is sufficient for this project. However, notice that all metaclasses, except for those that link information, inherit some properties from the common superclass `SynchronizedInformation`, which indicates that the information is synchronized on request of engineers (i.e., the information has been marked for synchronization in the respective engineering tool). Specifically, it is saved for every element which engineer performed the last modification, and when this modification happened. The metaclasses that link information within a type of development artifact do not inherit from `SynchronizedInformation` as they are generated automatically by tool adapters.

At the ontological level `O0`, the state of the DesignSpace after step 4 from the challenge problem in Section III is shown. Note that the relevant elements from each of the three involved development artifacts have been synchronized with the DesignSpace automatically by tool-adapters after they have been annotated by engineers in their respective engineering tools. For the source code, the three variable and the IO-port mapping from Fig. 2(d) are represented. For the EPlan P8 model, the three components that model the control signal of the motor conveyor and the two outputs of the Siemens PLC are represented. The links between the elements, which have been generated automatically by the respective tool adapters, are drawn as boxes with grey background. For the Excel sheet from Fig. 2(e), there are no annotated cells, meaning that nothing has to be synchronized with the DesignSpace.

B. Traceability

A major goal of VHA for this project was to allow for traceability between development artifacts so that after a change has been performed it is possible to easily identify other artifacts that might be affected and their associated engineers. After integrating all involved artifacts with the DesignSpace data integration framework, its traceability service can be used to establish traceability between (parts of) the different artifacts. While this may be done (semi-)automatically using state-of-the-art heuristics to identify the parts that are logically connected (e.g., a cell in a spreadsheet that is used for defining a constant in source code), the solution we developed relies on manual linking performed by engineers. For creating, updating, and deleting links between the different artifacts, we developed a standalone desktop application, the DesignSpace *Workbench*, which is based on the Eclipse Platform. This application visualizes any information present in the DesignSpace uniformly and traces can be established between arbitrary pieces of information. For example, one cell in an Excel sheet may

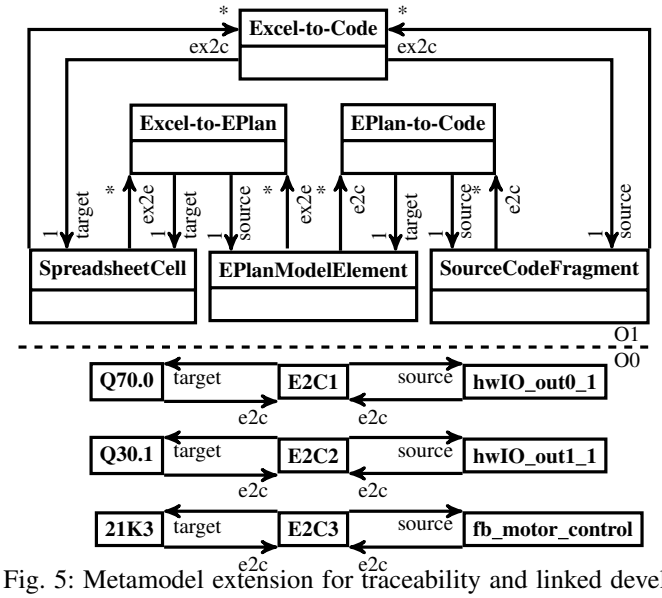


Fig. 5: Metamodel extension for traceability and linked development artifacts.

be linked to another cell in the same spreadsheet, to another cell in a different spreadsheet, to a constant or also a method in source code, or to an element of the electrical model. To create a link, an engineer simply has to specify the two pieces of information that are logically connected. Different kinds of links can be used to express additional meaning or to restrict the information that is eligible as source or target of the link. For example, for this project, in addition to the linking metaclasses already presented above that are responsible for linking information within individual development artifacts, we provide by default three kinds of links that may be used to link pieces of information that stem from different development artifacts: `Excel-to-Code`, `Excel-to-EPlan`, and `EPlan-to-Code`. These kinds of links simply restrict the source and target information to specific types. For instance, an `Excel-to-Code` link allows only cells from an Excel spreadsheet as source and only elements of source code as target. These three kinds of links are considered bidirectional, they just express that there exists a connection between the linked elements. In Fig. 5, the metamodel for traceability used in this project is depicted. Note that it contains one metaclass for each of the pre-defined types of traces at the ontological level `O1`. In Fig. 3, the creation of an `EPlan-to-Code` link is shown. Specifically, the left-hand side of the figure shows the EPlan model, whereas on the right-hand side the Eclipse IDE with Java source code is shown. The component named `Q70.0` in the EPlan model (selected and drawn in blue), which models the `Normal Output` IO-port of the `Siemens PLC` in Fig. 2(b), should be linked to the variable in source code that reflects this IO-port `hwIO_out0_1`. In the center of Fig. 3, the Workbench wizard for creating such a link is opened. In this wizard, the engineer has opened the information available in the DesignSpace of the EPlan model (right-hand side of the wizard) and the source code (left-hand side of the wizard). The engineer simply selects the DesignSpace representation of the IO-port (`Q70:0`) and the corresponding source code fragment representation (`hwIO_out0_1`) to establish the desired link. After linking each EPlan P8

model element to its corresponding source code fragment, the resulting DesignSpace representation is depicted in the bottom part (i.e., the ontological level `O0`) of Fig. 5.

Note that additional kinds of links can be defined by engineers freely and on-demand. For example, an engineer may want to use a "rational"-link to connect certain elements in the electrical model to a requirement, while another engineer may want to use an "equality"-link to express that a certain constant in the source code exactly reflects a value that is defined in a spreadsheet. Note that it is possible for engineers to express link semantics through additional link constraints. This allows, for instance, for different units in different artifacts. For example, a link may be enriched with a constraint that ensures correct translation between a metric value in a spreadsheet (e.g., 2,400 mm) to an imperial value in an electrical model (e.g., 94.48819 inches), or 2.4 kW to 2,400 W. The *Object Constraint Language (OCL)* is used for defining these constraints. Using OCL, more complex and sophisticated constraints can also be stated and validated. Next, we present in detail how constraints are written and validated.

C. Consistency Checking

As discussed in Section IV, the main objective of this project was to support consistency checking among VHA's major development artifacts. Above, we have seen how these artifacts are integrated with the DesignSpace platform and how they can be connected in the DesignSpace to establish traceability. Besides its obvious benefits, traceability is also a requirement for enabling the employed consistency checking approach. Specifically, our solution relies on the Model/Analyzer consistency checking framework [2], [6]. The Model/-Analyzer framework allows for highly efficient, incremental consistency checking of any information that is available in the DesignSpace. To enable incremental re-validation after changes of information, it reacts to changes in the data structure it checks. Therefore, a re-validation is triggered whenever the information available in the DesignSpace changes (i.e., whenever an engineer performs a change in his or her local tool and the change is synchronized with the DesignSpace). However, the re-validation works incrementally, meaning that only those consistency rules are re-validated that are potentially affected by the change. It has been shown that this incremental re-validation of consistency rules after changes to the checked data is typically performed within milliseconds. Thus, the information about changes regarding consistency between development artifacts can be provided instantly to engineers. All tasks regarding consistency checking (e.g., definition of consistency rules, or the notification about and the visualization of inconsistencies) are done in the Workbench tool we developed for this project.

Indeed, to check consistency among information from different artifacts, besides the links between these artifacts there must exist consistency rules that express their desired relation. These consistency rules for the Model/Analyzer are written in OCL, as already briefly discussed above. For each consistency rule, a *context* must be defined. This context defines which elements are checked by the rule. Specifically, the consistency rule is validated individually for each logical instantiation of the defined context (i.e., a rule context at the ontological level `L1` means that each instantiation of the rule context at `L0` is validated). When using the Model/Analyzer and the DesignSpace platform, any piece of information available in the DesignSpace may be used as a consistency rule's context. For instance, a consistency rule with the context `Class::Java` means that the rule is validated for every occurence of a Java class in the DesignSpace. However, let us focus on four consistency rules actually used in this project.

```
1 [Context: EPlan-to-Code]
2 self.source.e2c->contains(self) and
3 self.target.e2c->contains(self)
```

Listing 1: Consistency Rule for `EPlan-to-Code` Link

```
1 [Context: SpreadsheetLink]
2 self.source<>self.target and
3 self.source.link->contains(self) and
4 self.target.link->contains(self)
```

Listing 2: Consistency Rule for `SpreadsheetLink`

1) EPlan-to-Code Link: Let us first show how consistency rules can be used to ensure that the traceability features discussed above are used correctly. For links between EPlan P8 models and source code fragments, we want to make sure that the links are also accessible from the `EPlanModelElement`s and the `SourceCodeFragment`s they are linking. As an example, consider the link created between the electrical component `Q70.0` and the source code fragment `hwIO_out0_1` in Fig. 3. This particular link should be navigable from both `Q70.0` and `hwIO_out0_1` by using the reference named `ECLink` of the respective DesignSpace representations of the two components (i.e., `(Q70.0).ECLink` and `(hwIO_out0_1).ECLink` should point to collections of `EPlan-to-Code` links that the respective elements are part of, and these collections should both include the specific link). This can be ensured with the consistency rule shown in Listing 1. The context of this consisteny rule is `EPlan-to-Code`. Therefore, the consistency rule is validated individually for each occurence of such a link, including the `EPlan-to-Code` link between `Q70.0` and `hwIO_out0_1`.

2) SpreadsheetLink and SourceCodeLink: Similarly to the `EPlan-to-Code` links, we want to ensure that links connecting different spreadsheet cells or source code fragments are used correctly. As shown in the metamodel for linking in Fig. 5, a spreadsheet cell's representation in the DesignSpace may be linked to another cell by a `SpreadsheetLink`. However, it is not desired that a cell can be linked to itself. Therefore, the consistency rule in Listing 2 does not only check that the link is also accessible from the cells it connects, but also that it really connects distinct cells.

The same checking can be done for links between source code fragments. The corresponding consistency rule is equivalent to the one shown in Listing 2 but with `SourceCodeLink` as context.

3) EPlanModelLink: Finally, for the sake of completeness, links between elements in EPlan P8 models should also only exist between distinct elements, which is ensured with the consistency rule shown in Listing 3.

```
1  [Context: EPlanModelLink]
2  self.in<>self.out and
3  self.in.link->contains(self) and
4  self.out.link->contains(self)
```

Listing 3: Consistency Rule for `EPlanModelLink`

```
1  [Context: EPlanModelLink]
2  (self.in.e2c->size()>0 and self.out.e2c->size()>0) implies
3   self.in.e2c->exists(
4    x:EPlan-to-Code|self.out.e2c->exists(
5     y:EPlan-to-Code|x.source.link->exists(
6      z:SourceCodeLink|
7       ((z.source=x.source and z.target=y.source) or
8       (z.source=y.source and z.target=x.source)))))
```

Listing 4: Advanced Semantics Consistency Rule for `EPlanModelLink`

Now that we have described consistency rules that ensure correct traceability, let us move on to consistency rules for ensuring correct relations between different development artifacts, such as EPlan P8 models and Java source code. Specifically, we want to make sure that whenever two components in an electrical model are linked, their respective representations in source code are also linked. The corresponding consistency rule, which is instantiated for every `EPlanModelLink`, is shown in Listing 4. Indeed, this can only be checked if both of the linked model elements (i.e., the link's `in` and `out EPlanModelElements`) are represented by a source code fragment (i.e., the target of a `EPlan-to-Code` link). This can be ensured by checking whether both elements are part of at least one `EPlan-to-Code` link (see line 2 of Listing 4). If that is the case, then both linked `EPlanModelElements` must be part of `EPlan-to-Code` links to `SourceCodeFragments` that are connected by a `SourceCodeLink`. Those `EPlan-to-Source` link for the `in` and `out EPlanModelElements` are named `x` and `y`, respectively, in the consistency rule (see lines 4–5 of Listing 4). The existence of an appropriate `SourceModelLink` between the source code fragments is checked in lines 6–8 of Listing 4. Since `SourceCodeLink` should only express a connection between source code fragments without a prescribed direction but `SourceCodeLinks` have a source and a target `SourceCodeFragment`, it is necessary to check for the two possible directions (see lines 7–8 in Listing 4). Now that we have presented how development artifacts are integrated, which kinds of links can be used to establish traceability between different development artifacts, and how this traceability can be used to define consistency rules, we next have a look at how this can be applied to the challenge problem from Section III.

VI. Application to Challenge Problem

We will now revisit the challenge problem from Section III and discuss how our solution is applied during the most important steps.

A. After Step 3

We start with step 3. After this step, engineers have defined the initial versions of the EPlan P8 model and the corresponding source code, as depicted in Fig. 2(c) and Fig. 2(d), respectively. These two development artifacts have already been annotated by the engineers and the corresponding DesignSpace representations have been created automatically by the tool-adapters. Moreover, engineers have established traceability between the development artifacts, as shown by the presence of `EPlan-to-Code` links. Figure 6 shows the resulting DesignSpace representation. At this point, there are no inconsistencies. Notice that for the `in` and `out` elements of the `EPlanModelLink`, the corresponding source code fragments are also connected via a `SourceCodeLink`. Therefore, at this stage of development all involved artifacts are consistent.

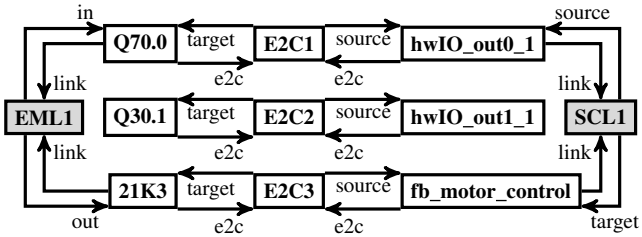


Fig. 6: Development artifact representation in the DesignSpace after step 3.

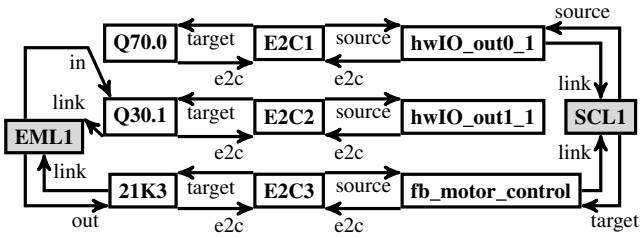


Fig. 7: Development artifact representation in DesignSpace after step 5.

B. After Step 5

After step 5, the EPlan P8 model has been changed by the electrical engineer to reflect the requests issued by the safety engineer in step 4. The DesignSpace representation of the electrical model has been updated automatically by the EPlan P8 tool-adapter, the result of this update is depicted in Fig. 7. At this point, the consistency rule from Listing 4 is re-validated for the link that now connects the components `21K3` and `Q30.1` (instead of `Q70.0`). However, since the source code has not been changed, there is no link in the source code representation that connects the corresponding source code fragments. Therefore, an inconsistency is detected and visualized right after the change of the electrical model. The information presented to the engineers shows exactly that the reason for the inconsistency is that the mapping in the source code is not conforming to the mapping in the EPlan P8 model. Moreover, the Workbench also lets the electrical engineer, who performed the change of the electrical model, identify the software engineer(s) responsible for the involved source code fragments. Of course, the software engineers will also be informed in the Workbench about the inconsistency. Therefore, not only the inconsistency is detected, but more specific information about the impact of the changed electrical model is available.

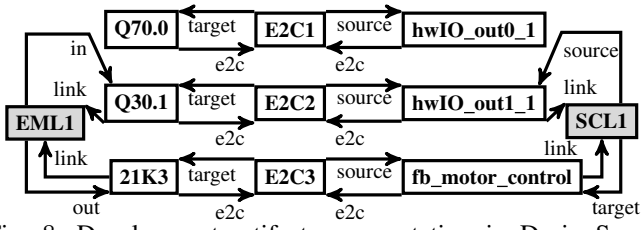


Fig. 8: Development artifact representation in DesignSpace after step 6.

C. After Step 6

After reacting to the inconsistency that was detected after step 5 by performing step 6, the inconsistency is removed by the software engineer by changing the mapping between source code fragments. In Fig. 8, the resulting representation of development artifacts in the DesignSpace is depicted. Notice that the `SourceCodeLink SCL1` has changed compared to Fig. 7. Specifically, it now uses as source the safety output of the Siemens PLC, `hwIO_out1_1`. When re-validating the consistency rule from Listing 4 for the `EPlanModelLink EML1`, the consistency checker detects that now a `SourceCodeLink` exists that connects right source code fragments (i.e., those that represent the `EPlanModelElement`s that are linked by `EML1`). Again, the change is processed immediately by the consistency checker and the engineer is informed immediately that the change actually removed the inconsistency.

VII. Discussion

Now that we have presented the solution developed in this project and its application to the challenge problem, let us revisit the goals and requirements stated in Section IV and discuss whether these are met by the solution. Moreover, in this section we will discuss the major challenges that were encountered during the project.

A. Goals

1) Traceability: As far as traceability is concerned, our solution provides the means to establish traceability between the integrated artifacts (i.e., spreadsheets, electrical models, and source code). The DesignSpace workbench can be used not only to establish traces of the pre-defined types, but also new types of traces with custom semantics can be defined by engineers. Using our solution, engineers at VHA can easily create and maintain traceability between different development artifacts as standard tasks that are added to their established development process. Engineers are expected to incrementally build traceability information by creating traces each time they make use of existing information they received from other engineers. While managers at VHA and Flanders Make are confident that this practice can be adopted by engineers, they believe that more support for engineers is still necessary.

2) Consistency Checking: For consistency checking, we have developed a set of consistency rules that are validated automatically and incrementally by the employed Model/Analyzer consistency checking framework. VHA believes that these consistency rules are sufficient to check electrical models and source code for consistency, which was the main goal of this project. Moreover, feedback about inconsistencies is provided instantly (or at any desired point in time) to engineers. However, more consistency rules may be added to extend the current set of consistency rules to also cover, for instance, spreadsheet data.

3) Change Impact Analysis: Based on traceability and consistency checking, our solution provides change impact analysis that allows engineers to quickly identify after performed changes which development artifacts may be (or are) affected and which engineers are responsible for those development artifacts. Therefore, our solution provides the information desired by VHA. Overall, the developed solution helps VHA to realize their goals of establishing traceability and consistency checking in order to improve the efficiency of their development process.

B. Requirements

The requirements stated in Section IV are met by the developed solution. In particular, the stated tools have been integrated with tool-adapters. Moreover, change impact analysis is now possible and dependencies between development artifacts are managed and visualized by the DesignSpace Workbench application.

C. Challenges and Lessons Learned

One factor that has been underestimated in the early phases of the project was the development of tool-adapters. Developing a tool-adapter for a commercial, closed-source tool with a defined API required us not only to familiarize ourselves with that API, but it also required all involved stakeholders to analyse and understand exactly which information is actually available to plug-ins or extensions, and in which form. Only after an in-depth analysis of the available information, it is possible to successfully develop a tool-adapter that is capable of synchronizing the relevant tool information with the DesignSpace. For example, the EPlan P8 tool-adapter developed in this project has to automatically discover the links between two or more components that are synchronized with the DesignSpace, as it is not possible for a user of the tool to mark these links. Moreover, it can be an additional challenge to observe a tool's internal data structure for incremental changes in order to synchronize its data live with the DesignSpace (or any other cloud solution, for that matter). However, typically tools allow at least for a synchronization at some events during their typical workflow, for example when the user saves changes. Additionally there are significant differences between tools with respect to their extensibility and the expressiveness of information available for third-party plug-ins or extensions. However, for this project we were able to access the relevant information in all three integrated tools. Moreover, it has been shown that in general most commercial application can be extended appropriately [7].

Overall, the results of the project show that even the basic forms of traceability and consistency checking that our solutions provides to VHA may help to improve the efficiency of a development process significantly. Engineers and managers at VHA were generally perceiving the use of the provided tools as straightforward and easy to learn. While only time can tell whether the practice of establishing and managing traces is actually performed continuously by engineers, the involved

stakeholders agree that this additional task does not impose a significant amount of additional work that would severely impact their existing workflow.

VIII. Related Work

Indeed, the topics of traceability and consistency checking have been discussed extensively in literature and they have also been addressed by commercial tool vendors. In this section, we will discuss those approaches and tools that are closest to the solution developed in this project and highlight the differences between existing approaches and tools and the results of this project. For consistency checking, various approaches have been proposed by academics and several commercial tools exist that perform some sort of consistency checking (e.g., [6], [8]–[12]; [13]). Unfortunately, to the best of our knowledge, there is no commercial or academic tool available that supports consistency checking of EPlan P8 models, let alone support for checking consistency between EPlan P8 models and other development artifacts. For academic approaches, as with commercial tools, they do not provide out-of-the-box support for EPlan P8 models, meaning that an adaptation of any existing approach was necessary. We opted for employing the Model/Analyzer consistency checker [2], [6] as it has been shown that it scales well and provides instant feedback about inconsistencies, even for large-scale industrial models [14]. Moreover, the Model/Analyzer not only provides engineers with information about inconsistencies, but it also provides features that automatically, based on individual inconsistencies, provide engineers with suggestions on how to repair them [15]. Of course, there also exist other technologies that allow for automatic model repairs (e.g., [9]). However, such approaches typically select and execute repairs automatically. Discussions with VHA and Flanders Make showed that such automatically performed changes are not desired as engineers wish to have ultimate control about the development artifacts and do not want the artifacts to be changed by fully automated technologies. Thus, the use of the Model/Analyzer, which only informs about inconsistencies and, if desired, possible repair options, is a valid choice. For traceability, different approaches have been proposed in the past, mainly focusing on automatic creation and management of traces [16], and tools often support some level of traceability, for example through change tracking (e.g., Excel). However, discussions with VHA and Flanders Make showed that automatic discovery of traces was not desired as there is always a chance of false positives that must be investigated. Therefore, we decided to rely on manually managed traces. Relying solely on different tools' traceability features was indeed not an option as none of the involved tools allowed to specify any kind of connection to development artifacts of other tools.

IX. Conclusions and Future Work

In this paper, we presented the results of a collaboration between Van Hoecke Automation, Flanders Make, and JKU's ISSE to establish traceability, consistency checking, and impact analysis in a company that focuses on the development of automation solutions. During the project, a software solution was developed that effectively helpes the company to improve its development process by providing traceability capabilities for different development artifacts and instant consistency checking between them. We found that one of the major challenges was the adaptation of existing technologies to support the domain-specific commercial tools used by the company. For future work, we plan to investigate possibilities of making the integration of such tools easier so that the effort for establishing traceability and consistency checking technologies can be reduced and the acceptance and application of these technogies increases.

X. Acknowledgements

This research was supported by Van Hoecke Automation, Flanders Make, the Austrian Science Fund (FWF): P25289-N15, and the Austrian Center of Competence in Mechatronics (ACCM): C210101.

References

[1] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, and A. Egyed, "DesignSpace – An Infrastructure for Multi-User/Multi-Tool Engineering," in *SAC*, pp. 1486–1491, 2015.

[2] A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE*, pp. 347–348, 2010.

[3] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Softw. Eng.*, vol. 37, pp. 188–204, March 2011.

[4] M. Fahad, N. Moalla, and A. Bouras, "Towards ensuring satisfiability of merged ontology," in *ICCS*, pp. 2216–2225, 2011.

[5] K. Kotis, G. A. Vouros, and K. Stergiou, "Towards automatic merging of domain ontologies: The hcone-merge approach," *J. Web Sem.*, vol. 4, no. 1, pp. 60–79, 2006.

[6] A. Egyed, "Instant consistency checking for the UML," in *ICSE*, pp. 381–390, 2006.

[7] A. Egyed and R. Balzer, "Integrating cots software into systems through instrumentation and reasoning," *Autom. Softw. Eng.*, vol. 13, no. 1, pp. 41–64, 2006.

[8] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik, "Global consistency checking of distributed models with tremer+," in *ICSE*, pp. 815–818, 2008.

[9] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE*, pp. 455–464, 2003.

[10] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards automated inconsistency handling in design models," in *CAiSE*, pp. 348–362, 2010.

[11] S. Easterbrook and B. Nuseibeh, "Using viewpoints for inconsistency management," *Software Engineering Journal*, vol. 11, no. 1, pp. 31 –43, 1996.

[12] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *ICSE*, pp. 511–520, 2008.

[13] IBM, "Rational Software Architect." http://www-01.ibm.com/software/rational/products/swarchitect/.

[14] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *MoDELS*, pp. 202–218, 2012.

[15] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of metamodels and models through consistent change propagation," *Journal of Systems and Software*, vol. 111, pp. 281–297, 2016.

[16] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *FOSE*, pp. 55–69, 2014.