Reactive Links Across Multi-Domain Engineering Models

Cosmina Cristina Rațiu
Johannes Kepler University
Linz, Austria
cosmina-cristina.ratiu@jku.at

Wesley K. G. Assunção
Johannes Kepler University
Linz, Austria
wesley.assuncao@jku.at

Rainer Haas
Linz Center of Mechatronics GmbH
Linz, Austria
rainer.haas@lcm.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

As the engineering world moves towards collaborative model-driven development, it is becoming increasingly difficult to keep all model artifacts synchronized and consistent across a myriad of tools and domains. The existing literature proposes a variety of solutions, from passive trace links to computing change propagation paths. However, these solutions require manual propagation and the use of a limited set of tools, while also lacking the efficiency and granularity required during the development of complex systems. To overcome these limitations, this paper proposes a solution based on reactive propagation links between property values across multi-domain models managed in different tools. As opposed to the traditional passive links, the propagation links automatically react to changes during engineering to assure the synchronization and consistency of the models. The feasibility and performance of our solution were evaluated in two practical scenarios. We identified a set of change propagation cases, all of which could be resolved using our solution, while also rendering a great improvement in terms of efficiency as compared to manual propagation. The contribution of our solution to the state of the practice is to enhance the engineering process by reducing the burden of manually keeping models synchronized, eliminating inconsistencies that can be originated in artifacts managed in a variety of tool from different domains.

ACM Reference Format:
Cosmina Cristina Rațiu, Wesley K. G. Assunção, Rainer Haas, and Alexander Egyed. 2022. Reactive Links Across Multi-Domain Engineering Models. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552446

1 INTRODUCTION

The basis of each engineering process are design decisions based on requirements and their realization in implementation artifacts such as models, prototypes, and source code [33]. However, engineering artifacts are living documents, that are constantly refined, changed, and updated [10, 15, 24, 31]. During the development time, engineers learn more about the project, different off-the-shelf components are selected, and progressively complex prototypes are presented to the clients [3, 31]. With each of these steps, the constraints and dependencies between the artifacts, e.g., model elements, are updated and consistency rules are defined [17, 44]. In all phases of the engineering process, the artifacts of the system in development have to be consistent with each other and have to adhere to the constraints imposed by the requirements and other related implementation artifacts [8, 39]. This is not an easy task by default, but becomes even more of a challenge in the context of collaborative, multi-domain and model-driven engineering [6, 7, 28, 34].

Nowadays, any complex system, independent of its size, is developed by multiple teams [20, 26, 38]. Each team focuses on one aspect of the system, using a separate tool, concerning about domain-specific properties, and creating different, but related, artifacts [30]. Nonetheless, all these artifacts characterize the same system and, therefore, need to correspond and be synchronized. If the specifications are changed to require new property values in one tool, this change has to be propagated over every dependency affected. Moreover, if this results in new affected dependencies, these must also be propagated.

The literature has explored the issue of linking artifacts from multiple perspectives, offering different solutions which aid the engineers in the collaboration process. Creating trace links [5, 11, 27, 35, 37] between related artifacts across tools helps the management of the dependencies between models, but does not propagate the changes automatically, therefore, being only *passive links*. Setting consistency rules [1, 4, 42] can highlight inconsistent data, but often cannot provide the granularity necessary for all use cases, even when paired with traceability. Change propagation algorithms [2, 22, 32, 39, 43], whether they are determining the ideal propagation paths or predicting the impact of a change, still require manual propagation and often slow down the development process through their high computation times. The few solutions which can provide automatic change propagation across linked artifacts [13, 14, 16, 21, 29, 36] require the use of a certain, very limited array of tools, and often cannot handle multi-tool and multi-domain systems. These limitations hamper proper engineering of complex systems by omitting dependencies or hiding potential inconsistencies, directly impacting the quality of final products.

The goal of our paper is to address the above limitations by proposing a solution that combines traceability [9], consistency checking [19], and change propagation [23], with the addition of finer granularity. Our solution relies on "*reactive links*" to automatically propagate fine-grained changes, namely at the level of model properties, across multi-domain artifacts of complex systems. These links allow connecting the values of related properties in different artifacts within a developing system. When a change in the system affects a linked property, our solution instantly applies its impact on all related artifacts, as configured by the users. These

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9466-6/22/10.
https://doi.org/10.1145/3550355.3552446

reactive links avoid unnoticed inconsistencies that might linger in the system until they create severe faults.

We evaluate our solution by applying it in two engineering scenarios,[1] which capture a wide range of domains, tools and change propagation cases. We successfully observed the feasibility and performance of the reactive links in both scenarios, including a significant improvement in terms of maintaining constraints and consistency among artifacts. Additionally, we have identified corner cases in which the links can be expanded to significantly enhance the collaborative model-driven engineering process. Finally, we found a number of improvements to be made in the future, as well as a list of possible ways to enhance and specialize the solution towards different specific use cases.

The main contribution of our solution is to significantly reduce the overhead required to maintain the dependencies and consistency between different models in collaborative engineering. Our solution can help engineers to:

- Reduce the likelihood of errors appearing during the modeling process, e.g., validation of the product, by highlighting any dependency violation between model artifacts.
- Reduce the risk of models using outdated data or overlooking inconsistencies by immediately propagating changes as soon as they appear in any of the artifacts.

The remainder of this paper is organized in sections. Section 2 introduces an example of risks involved in synchronizing related artifacts, followed by the description of the proposed solution in Section 3. Section 4 presents the study design and the research questions we address with the evaluation using two engineering scenarios. Sections 5 and 6 discuss the results of the evaluation and the threats to validity, and Section 7 describes the solutions found in the literature. Finally, Section 8 draws the final conclusion on the suitability of our solution for the problem at hand.

2 MOTIVATIONAL EXAMPLE

In the following, we present an illustrative example, which was extracted from one of the engineering scenarios we consider as part of the evaluation. Let us assume the development of a hydraulic actuator. This process consists of three teams working in collaboration: (i) a team of *requirement engineers* using a requirements tool, (ii) a team of *physicists* using a complex calculator to determine the parameter values that best describe the system, and (iii) a team of *graphical designers* who model prototypes of the system iterations using a graphical modeling tool.

The hydraulic actuator is composed of a cylinder with a piston, which is connected to a pump. For this example, we focus only on the cylinder. A requirement specifies that the pressure within the cylinder shall not exceed the atmospheric pressure, which is *160 bar*. This is a very important threshold for the safe and correct operation of the piston. The physicist team use a calculator worksheet dedicated to the cylinder for setting the pressure as an input, also inserting other significant data, which results in the diameter of the piston cylinder. This team is looking for the minimum cylinder diameter for which the actuator still works safely, thus they insert the threshold value for the pressure. The resulting diameter is then used by the graphical modeling team to update their design, and in turn, communicate to the client the threshold values to be considered when choosing off-the-shelf parts.

[1] From Linz Center of Mechatronics GmbH, Austria. https://www.lcm.at/en/

2.1 Problem Statement

In an ideal situation, the three models the development is based on would be completely synchronized. However, this is not a common situation in practice, especially when we consider the collaboration between different teams with different levels of insight in each model. Human mistakes are not unlikely, and if each team is responsible for only one tool and one model, the constraint violations and inconsistencies might be easy to miss.

Let us assume, for example, that a mistake happens when the required maximum piston pressure is introduced, resulting in a value of *170 bar* in the calculator artifact. This value goes on to determine a piston diameter, which would actually result in an unsafe system. Without being aware of the requirement constraint (maximum 160 bar), engineers introduce this value in the graphical design. However, the graphical modeling team does not have the insight or responsibility to determine whether the values resulted from the computation are correct, nor does the requirements engineering team. This incorrect value may reside in the system until the quality assurance team notices the issue late in the development process and reports it. Correcting the mistake requires the collaboration of all the teams involved and additional reviews. This lengthens the process, adding to the costs and effort required. Left undetected, the mistake results in the client acquiring improper and unsafe components based on incorrect information.

2.2 The Limitations of the Existing Solutions

Next we briefly describe how solutions existing in the literature, which are presented in details in related work (Section 7), would be applied to the problem. Additionally, we highlight the limitation of such existing solutions. While each one would solve part of the problem, none of them addresses the whole issue, and each comes with its own set of disadvantages.

2.2.1 Trace Links. Two trace links can be used to connect (i) the pressure in the requirement to the pressure in the computation and (ii) the computation result to the cylinder diameter in the graphical design. But there are two major drawbacks to this approach:

- The role of the two relationships represented here are very different. In one case, namely (i), the goal is to keep the value consistent to a *threshold*, while in the other case (ii), it is expected that the values of the properties are *equal*. The engineers then have a choice between creating two different trace matrices only for these cases, or selecting very general trace types that do not provide all the information they need.
- The traces are, in essence, passive links. They cannot propagate information or check the constraints, they only mark the relationships between artifacts. The assurance that the constraints are met still falls of the engineers.

2.2.2 Consistency Rules. Engineers may choose to set consistency rules. Using consistency rules firstly requires a way to connect the artifacts across the tool boundaries. In systems that can be reduced to one metamodel, this is not a concern. When connecting artifacts across models, consistency checking can be paired with

traceability, which will maintain most of the downsides discussed before, especially the lack of granularity. Trace links are defined at artifact level, for example, connecting requirements to the calculator artifacts that use them. But there is only one requirement that specifies the maximum piston pressure. Setting a consistency rule using traces will result in latency and redundancy, as every trace is checked for a parameter that is only specified once, potentially resulting in falsely marking as inconsistent all the requirements that do not specify a maximum piston pressure.

2.2.3 Change Propagation. Let us assume engineers have a change propagation algorithm set up to deal with the system under development. This in itself is a significant supposition, considering as most change propagation algorithms struggle to deal with multi-domain data. In addition to that, there are still some main issues that come with these algorithms:

- Change propagation algorithms require extensive computation resources and time. They take an overview of the system, usually in the form of a matrix with all the dependencies between artifacts, and parse it in order to simulate the propagation. Even in small systems, such as the hydraulic actuator, the number of artifacts to be featured in this matrix/parsing is significant.
- The change propagation is usually not done automatically. Propagation algorithms can suggest ways to deal with constraints and fix inconsistencies, but engineers must to manually apply the necessary changes. This adds to the overall time required for the system to return to a consistent state. Additionally, this process does not fully address the risk of human mistakes.

2.2.4 Automatic Knowledge Propagation. Links that automatically propagate changes are, in theory, the most viable solution to solve inconsistencies between the computation results and the graphical design. However, in practice, the solutions proposed in this sense are domain-specific. For example, most of the solutions are only available for systems entirely described using Unified Modeling Language (UML), which is clearly not the case in our example. The few solutions that do not rely on UML, are also targeted towards a very narrow domain and cannot address the combination of artifacts of our example, namely requirements, computations, and graphical elements.

The existing propagation links also have the drawback of not addressing constraint violations. They can, in a suitable system, maintain the consistency of the values that have to be equal, but they have no way of dealing with constraints such as the piston pressure requirement in our example.

3 PROPOSED SOLUTION

The aim of our solution is to address the main downsides of the approaches discussed in Section 2.2, while also selecting and enhancing the benefits of each one.

3.1 Artifacts Representation

As mentioned in Section 7, most knowledge propagation solutions require a UML representation. However, our solution is designed to work in a multi-domain and multi-tool scenario. For that, we

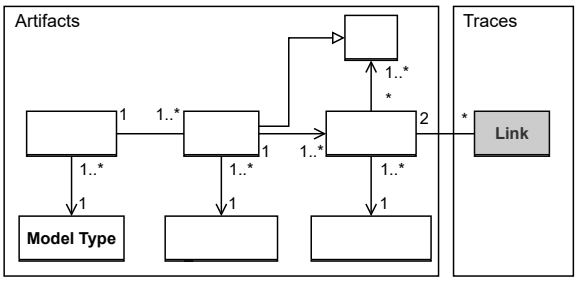


Figure 1: Representation of multi-domain models and links

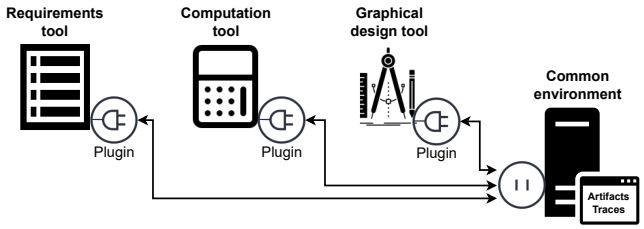


Figure 2: Common environment with multi-tools and plug-ins

defined a generic representation of artifacts, i.e., model elements, in which we can deal with arbitrary types of model and artifact.

Figure 1, on the left side, presents an overview of the metamodel defined to represent the artifacts, whereas the right side presents the links added by our solution. In this representation, a model is of a model type and consists of model elements. These elements have a type and properties. The properties of an element have a type and values. Elements are a subclass of values, as a model element can also be a value.

In the following, we assume that the artifacts represented according to this metamodel are stored in a *common environment*, as illustrated in Figure 2. This common environment is responsible for receiving call and trigger operations, each time a property is changed in the host tool, e.g., graphical design tool. For that, we also assume that the tools in different domains have a plugin that enable them to communicate to the common environment.

3.2 Linking Granularity

Trace links, as addressed in Section 2, make connections at model level. However, in most cases, the target information of the link is stored at the level of property values. Therefore, changes to other properties, which are irrelevant for the links, would still trigger the propagation or consistency checking even when they have no real impact on the linked details.

To address this lack of granularity, our solution allows linking specific properties that have a correspondence. For example, if we consider the piston pressure from our illustrative example (Section 2), linking the requirement to the computation directly would also connect many redundant elements, and most likely unrelated information. Instead, our solution enable engineers to link only the

property that stores the constraint value (160 bar) to the exact property in the computation. Changing the value of one of these two properties triggers a link execution, while any other change to the artifacts is not slowed down by executing irrelevant propagation links. In Figure 1, on the right side, we show the link associate with two properties. More details of the implementation of the links are described in Section 4.

3.3 Propagation Operators

With the properties linked, we must specify the operation to be performed when a relevant change happens. One major disadvantage of the existing solutions is that they are specialized in one specific operation, whether only propagating values directly, or flagging constraint violations for the engineers to later resolve manually.

Our solution encompasses both use cases in one mechanism. As such, when setting a link, the user selects the operator to be applied during when the property changes. An *operator* correspond to which action automatically happens with the linked properties. Alternatively, some operators can check whether a requirement constraint or a consistency rule is violated. Table 1 presents the link operators that can be applied in our solution.

Table 1: Operator to define the link actions

Operator	Name	Description
=	Assignment	Propagates the change to the linked property value
==	Equals	Checks the equality of the linked values
! =	Not equals	Checks the inequality of the linked values
<	Less than	Checks that the value is less than its counterpart
<=	Less than or Equal	Checks that the value is less or equal to its counterpart
>	Greater than	Checks that the value is greater than its counterpart
>=	Greater than or Equal	Checks that the value is greater or equal to its counterpart

3.4 Propagation Direction

A further point we noticed while designing the link operations is that, in many cases, the execution depends on which end of the link was changed. Recalling the hydraulic actuator in Section 2, we can select the link connecting the piston cylinder diameter in the computation and in the graphical model. The engineers expect that the value in the graphical design will always correspond to the computation, so they select an assignment operator (=) when creating the link. However, the piston diameter in the computation is a result and can only be changed if the input changes.

The links should, therefore, hold the knowledge about the *direction* of propagation. This means that the link will react depending on the origin of the change. In the previous example, a change originated in the computation is propagated, while a change in the graphical design result in inconsistency between artifacts. Table 2 present the three directions covered by our solution.

3.5 Additional Transformations During the Propagation

While many use cases work well with nothing more than direct propagation or constraint checking, there are use cases which require some additional transformations. For example, the hydraulic actuator requirements have a constraint for the safety against rod buckling. The constraint states that the hydraulic force shall be less than twice the buckling force, both of which are results in of different calculator worksheets. Addressing this constraint with a simple link would require an intermediate property value which computes the double of the buckling force. Instead, our linking solution allows engineers to specify *simple transformations* to be applied during the execution of the link. In this case, engineers can add the "2*" operation during the setting of the link, and the operation would be performed after each relevant change.

Table 2: Direction of the link operations

Name	Description
Bidirectional	The links are triggered and evaluated identically for changes in both properties
Unidirectional	The changes are propagated in one direction of the link, and ignored in the opposite direction
Asymmetrical	The link evaluation is different depending on the changed property value

3.6 Propagation Triggers

Finally, a remaining question is when exactly the link actions are triggered. Our proposed solution supports two kinds of *triggers*, as presented in Table 3. The *manual* propagation is the least intrusive, but requires more user interaction and care. The *automatic* propagation requires no user interaction after the creation, and allows for chain propagation, where one link evaluation triggers another to fully propagate the impact of a change. Each of both options has advantages and disadvantages, and they should be addressed and balanced regarding the engineer goals, types of artifact, criticality of the requirement constraints or consistency rules, and implementation details.

Table 3: Triggers to define when the link are executed

Name	Description
Automatic	Whenever a property value changes, all the links with these triggers are automatically executed
Manual	The user decides when to manually trigger the execution of the links

4 STUDY DESIGN

Our study aims to evaluate the proposed solution in terms of feasibility and performance. In the following, we present the research questions, two different engineering scenarios, and the implementation aspects of our solution.

4.1 Research Question

The goal of this evaluation study is to answer two research questions (RQs):

RQ1: Feasibility - Does the proposed solution cover all linking cases of our two multi-domain and multi-tool scenarios

and keep artifacts synchronized? We focus on exploring two scenarios in order to evaluate the behavior of our solution when dealing with a wide variety of constraints and consistency rules between different models. We observe the change propagation as part of two scenarios and check the correctness of the result.

RQ2: Performance - How does the execution time and accuracy of the reactive links compare to manual propagation? For this, we observe and measure the performance of the links, with a focus on execution time. Our focus falls on whether the reactive links provide a significant improvement as compared to having the developers propagate changes happens. In order to determine this, we will measure the execution time of multiple linked propagation cases found in our engineering scenarios.

4.2 Subject Systems

For evaluating our solution, we have considered two separate engineering scenarios, which are presented in Table 4. The scenarios *Hydraulic actuator* and *Robotic arm* have three and five domains, i.e., different number of engineering tools, respectively. They were developed by our collaborators working for Linz Center of Mechatronics based on real-world projects they have contributed towards in the mechatronics' industry. The distribution of the artifacts is detailed in Table 4.

Table 4: Details of the scenarios

Scenario	# Domains	# Artifacts	# Properties	# Links
Hydraulic actuator	3	293	1719	17
Robotic arm	5	981	14836	53

The number of links (last column in Table 4), 70 in total, provide us the desired variety and complexity of types, constraints, and rules, enabling us to evaluate our solution. All the link requirements came from engineers or from client needs. Figure 3 presents details of the different domains and artifacts, as well as the number of reactive links between such artifacts. Requirements should be linked to mostly all other artifact domains. Physical computations involve multiple artifacts which are connected to each other. In the following, we describe each of these two scenarios in details.

4.2.1 Hydraulic Actuator. This scenario[2] consists of a piston within a hydraulic cylinder, a pump, and a valve system assuring the connection between the two main components. For developing this project, the engineers model the pump starting with the initial requirements. Next, the teams go through iterations of model refinement. Finally, the engineers check that the building blocks of the system follow the requirement constraints and the safety conditions. There are three teams that collaborate in the engineering process, each with a separate set of responsibilities and expertise, as follows:

- *Requirement engineers* using our requirements tool.[3] This tool allows them to specify and manage the requirements, in which they can specify quality measurements and constraints. In the context of DesignSpace, these additional constraints are stored as custom property values associated with the requirement they are part of.
- *Physicists* use TechCalc, a computation calculator that has default worksheets corresponding to the components of the system. Engineers select the input parameters in the worksheets and the calculator applies the appropriate formulas to compute the rest of the parameters to characterize the components.
- *Graphical designers* use SolidWorks to manage a virtual model of the system, visible on the right of Figure 3 (a) based on the data received from the other two teams.

The actuator development process will have four stages: (i) *initial setup*, the requirements are collected and the preliminary computations and graphical components are set up; (ii) *threshold computation*, the values of the constraints are used to determine the most extreme component measurements for which the system still works safely, as presented in our motivating example; (iii) *model development*, the teams model the system based on the requirements and the progressive feedback they receive from the client; (iv) *selection of the off-the-shelf components*, a multi-stage process in which the client selects off-the-shelf components for the project and communicates the characteristics to the teams, then engineers assure the components will interact correctly.

4.2.2 Robotic Arm. This project[4] consists of a factory setup composed of several robotic arms that should pick up an object from the respective receiving bays and transport it to the target locations. The development is based on a basic setup, which is then adapted to the factory requirements provided by the client. For the development of this product, the collaboration is more complex. While the hydraulic actuator consists only of hardware, the robotic arm also requires a software program to control its movement. Therefore, the teams consist of:

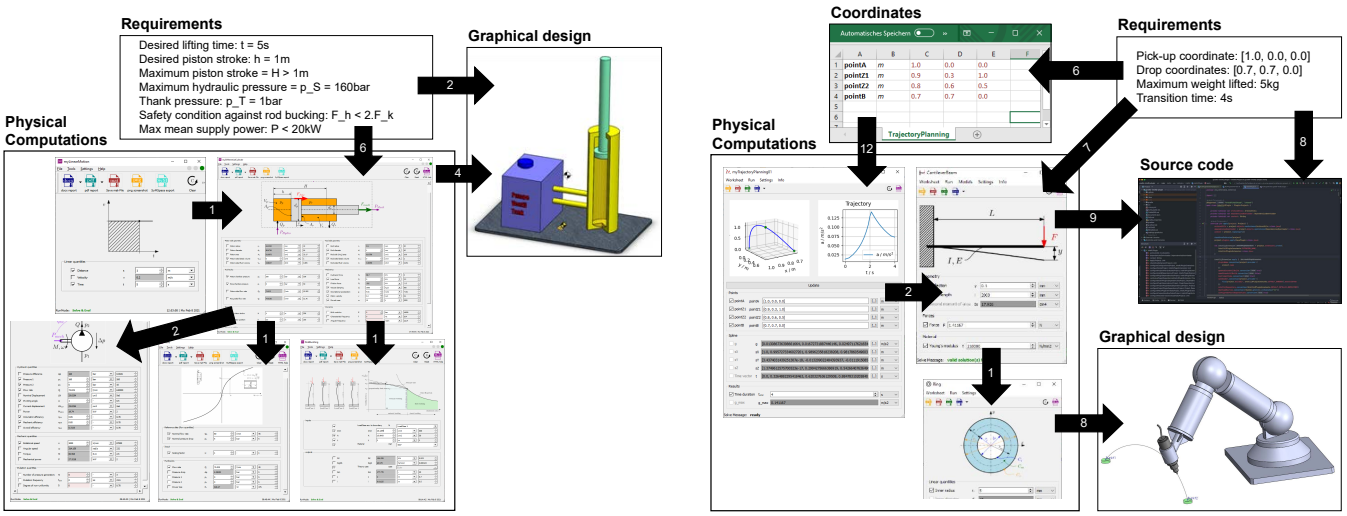
- *Requirement engineers* using the same requirements tools as the previous scenario.
- *Mathematicians* translate the real life setup of the bays and movement trajectory into relative coordinates respective to each robot base, stored in Microsoft Excel.
- *Physicists*, also using TechCalc, with a focus on the movement of the robot and the components necessary for the correct trajectory and functionality to be achieved.
- *Graphical designers* produce visualizations of the arm using SolidWorks.
- A team of *software engineers* who write and maintain the code which will control the movement. The code is written in Java, using the editor IntelliJ, and provides a basic setup of two different controllers. The most appropriate one is selected through configurations.

For this evaluation, we focused on the steps required for the basic setup to be specialized towards the specific requirements of a client. The models of the system were set up, including the correspondences between the artifacts. The requirements specify

[2]Video with a demonstration of this scenario is available at: https://youtu.be/4Z9qy9PM2PY

[3]A tool that has been developed within our institute

[4]Video with a demonstration of this scenario is available at: https://youtu.be/gdqROocDq30



(a) The hydraulic actuator scenario

(b) The robotic arm scenario

Figure 3: The scenarios, artifacts linked, and the number of links used for the evaluation.

the absolute coordinates of the loading bay and the drop point, as well as the robot base location and the requirements for its functioning for each robotic arm. The mathematicians then compute the coordinates from the perspective of the robot, which will influence the computations. The result of the computations specify the components required for the building of the robot, as well as the deflection of the arm, which is necessary to select the most appropriate controller code for each robot.

4.3 Implementation Aspects

In order to assess the feasibility of the solution, we present an implementation used to the evaluation study.

We start with the common environment that is a server where all the artifacts are stored. We have chosen to used DesignSpace[5] [12], which is a platform that allows the storage and reasoning of engineering knowledge in a multi-tool and multi-user environment. Any tool that can be connected to DesignSpace uploads its resulting models to the server, together with their respective metamodel. This data is kept synchronized and consistent, as specified by the engineers. DesignSpace follows the representation presented in Figure 1, which was the basis used to implement the links, presented on the right side of the same figure.

The connection between the common environment and the tools used in our two scenarios was done through the plugins and adapters. Each tool is provided with a custom-built adapter, responsible for translating between its internal metamodel to the DesignSpace data representation. Our solution is event-based, which means that any change to an artifact results in notifications being sent to the services on the server and the tools connected to it. The communication between the server and the client-side extension is done via gRPC [18]. This communication protocol uses `proto` requests and responses to send messages across tools, platforms and interfaces. The messages encode all the information necessary for the receiver to understand and apply the change in their local data.

For our solution, we have added a custom module to DesignSpace, which is focused on reactive links. These links are represented by specialized instances within the DesignSpace system. Linking instances have a consistent and unified model for all use cases. The instances are of the type `Link`, show in Figure 4. This figure also illustrates the link using our motivating example (Section 2).

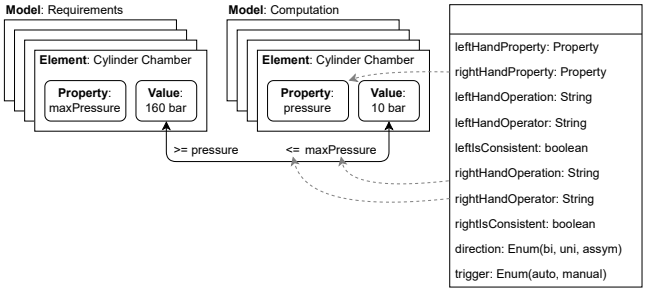


Figure 4: The structure of the link implementation

As mentioned in the solution description, the aim of the links is to assure granularity by linking property values directly. This is uniquely identified by the instances of **Property**. Additionally, `operator` and `operation` store the link properties, for each of the two ends of the link, together with `direction` and `trigger`, which completes the instances. The fields `IsConsistent` store the current status of each link instance.

Engineers can also specify simple transformations to be applied during the propagation. These are executed as a first step when a relevant change happens. For this process, we are using ARL [25], which is also used for unit transformations during the propagation.

[5]https://isse.jku.at/designspace/index.php/Main_Page

In order to set a new link, the engineers select and specify the instances and properties to be linked, as well as the operators and simple transformations. This information is exposed to be selected through a simple prototypical GUI. Upon the selection by the user, the data is transported through a specific `gRPC` call to the server, where a new link instance is created. Additionally, the server performs a recursive search through the link structure the two properties are already involved in, if any, to prevent cyclical linking. While we acknowledge this is not the only solution, cyclical links are likely to cause infinite cycle propagation in specific scenarios.

In terms of intrusiveness, DesignSpace is designed with collaboration as a main focus. Therefore, it provides a wide variety of features, such as workspace inheritance and committing model artifacts, to overcome collaboration issues and support the teams in sharing their work safely. As such, we find that the intrusiveness of automatic propagation is reduced significantly enough for the benefits of this propagation trigger, e.g., chain propagation, to outweigh its downsides.[6]

As stated before, the execution of a link is triggered by a change in one of the property values it connects. It goes through the following steps:

(1) The change notification is received and the changed property value is selected from it.
(2) The changed value is converted to the standard unit of measurement, if a unit is specified, e.g. a value in cm is converted to m.
(3) The operation is performed if one is specified.
(4) The operator is checked.
(5) For propagation, the value is converted to the unit of the opposite end of the link, and then set to the corresponding property, unless the values are already equal.
(6) For constraint checking, the opposite value is also converted to the standard unit, and then the two are compared. The result of the comparison is stored in the corresponding consistency flag.

5 RESULTS AND DISCUSSIONS

Based on the evaluation study, we were able to evaluate the feasibility and performance of our solution. We created reactive links for two engineering scenarios, linking all the related artifact properties based on requirements and consistency rules. In the following, we present each use case identified, together with a practical example from the scenarios. After that, we discuss the execution time required for different link distributions.

5.1 RQ1 - Feasibility

5.1.1 One-To-One Bidirectional Propagation. Firstly, we consider the most common and most simple use case, which involves connecting two property values, and bidirectionally propagating the changes from one to the other. One such situation can be seen in the hydraulic actuator scenario. The diameter of the piston cylinder has to be consistent between the computations (in TechCalc) and the graphical design (in SolidWorks). For this, we connected the two properties with a link, setting the operator in *both directions* to *assignment*. The result is that any change in the computation of the cylinder diameter, or in the graphical model, is instantly propagated to the other end of the link.

5.1.2 One-To-One Constraint Checking. A similar situation occurs when we are checking the consistency of a relation between two property values. In the example presented in Section 2, the requirement constraint on the piston chamber pressure can be linked to the value in the computation, with corresponding *comparison operators* on both directions of the link. After we set this link, if a developer accidentally sets the piston chamber pressure too high in the computation, the problem will be signaled in both of the tools which are involved. Similarly, if the requirement changes, the link will be triggered to assure the value already set in the computation still follows the requirement.

5.1.3 One-To-One Unidirectional or Asymmetrical Linking. There are cases in which the requirement specifies exact values to be used in other tools. In these cases, any change in the specification should be propagated to the other tools that use the corresponding value, while the specifications should not be changed via link propagation. For this case, our solution makes possible the use of a *unidirectional link*. The link is set just as described in the first example, however, we only set the assignment operator for the direction of the link that should propagate. The other direction will remain empty and be ignored. Alternatively, we can use *asymmetrical links* by setting the other direction operator to *equals*. In this case, a change in the specification will be propagated, and a change in the implementation that leads to a constraint violation will signal the inconsistency.

5.1.4 Links With Operations. Additional use cases involve situations where the values should not be propagated directly, but rather go through a simple transformation in the process. For example, in the robotic arm scenario, the minimum length of the robotic arm is computed in TechCalc. The actual arm length should actually be 20% bigger than this computed value. In this case, the links are set as previously, with the modification that we are specifying in each direction of the link, the *operation* necessary to compute the correct value, when necessary. Similarly, the type of the value is not always the same in both of the linked properties. In the same scenario, we used the robot arm length in the code of the software which run on the robotic arm. The arm length is a *floating point* value, while the Intellij adapter stores all the data in *text* format. Therefore, the link towards the code property will transform the value to a string. The opposite direction can have an operation which parses the string back to a number.

5.1.5 One-To-Many Constraint Checking. The cases discussed above only involve one link, but links can also interact with each other or can be combined to obtain more complex results. The first example of that is checking that a value is within an interval specified in the requirements. For this, we used in our solution *two comparison links*, each corresponding to a side of the interval. An example of this was captured in the hydraulic actuator scenario, where the value of the characteristic coefficient of the piston cylinder has to stay within a specified interval for the component used. This coefficient is a computation result, and when the value does not correspond to the specifications, the inconsistency will be marked. Additionally,

[6] A demonstrative video of the reactive links in a collaborative engineering scenario is available at: https://www.youtube.com/watch?v=hlfo9DOCCos

the developer will immediately see whether the value is too low or too high.

5.1.6 One-To-Many Propagation. Another way links can be used in our solution for more complex results is by creating one-to-many propagation paths. One example of this is in the hydraulic actuator scenario, where the length of the piston cylinder was specified in the requirements, and used in the TechCalc and SolidWorks models. We linked each of the properties in these two models to the requirement. When the requirement changed, the change was automatically propagated to both models in one single step.

5.1.7 Chain of Propagations. Lastly, having selected an automatic propagation, the links can also interact with each other and trigger each other. The simplest example of this is to link the requirement and the computation, and then the same computation to the graphical model, resulting in a similar situation as described before. However, a more practical example considers the case where one link triggers a computation, the result of which is part of another link. This use case is captured in the motivational example. A link connects the constraint threshold of the piston chamber pressure between the requirements and the computation. The resulting cylinder diameter in the computation is then linked to the corresponding graphical design artifact.

Now, let us assume that the client selects a component with a different threshold pressure and wants to see if it would still be safe to use. The specified threshold would be set to the new value, which would be instantly propagated to TechCalc. The TechCalc worksheet is also automatically recomputed, leading to a new result, which is propagated through the links to SolidWorks. This whole process happens almost instantly, and avoids the risk of human error when synchronizing the data.

> ***Answering RQ1:*** *All the identified linking cases in the two selected scenarios could be addressed using the reactive links, regardless of the domain of the artifacts. Unlike other existent solutions, the links can both propagate changes, by using the assignment operator, and check constraints. Additionally, they also allow simple transformations to be performed during the propagation, which were not covered by other knowledge propagation solutions.*

5.2 RQ2 - Performance

This question is whether the linking solution provides a significant advantage in terms of time as compared to manual synchronization. In order to evaluate this, we have selected the most common use cases identified in the two scenarios of our study. We have timed ten executions and computed the average, which will average out external factors such as the interference of other processes in the execution time.

To begin with, we must address the overhead of setting up the models and links, as this can appear to be an additional step in the development process. However, in practice, setting up the models using DesignSpace does not require any additional effort from the user apart from completing a simple connection form.

From the point of view of the links, even in the absence of our solution, the changes that occur during the development process have to be propagated across the dependencies within a model and between models. As such, these dependencies have to be identified and documented, preferably early on during the process. The reactive links can be set as soon as the dependencies are identified, which prevents cases in which a documented dependency would be lost or accidentally omitted.

In addition, setting a link requires the engineer to use the GUI prototype provided. This consists of a window allows the selection of the two instances to be linked, followed by the selection of the two properties to be linked. Additionally, the user selects the operators from a drop-down list, and have the option to specify the transformations in respective text boxes. When the selection is complete, the user concludes by pressing a button, upon which the respective link is created.

The first case evaluated is the *one-to-one bidirectional propagation*. After ten executions, we determined that the propagation across this type of link takes *0.045* milliseconds, which is already beyond the reaction time of any developer. We then considered the added impact of an *operation*, so we selected the case in which the robot arm length has to be propagated from the computation to the java code, and converted to string. This propagation took, on average, *0.053* milliseconds, the difference between this and the previous case being the application of the java `toString()` method.

Another common use case is the *chain propagation*. We have chosen to evaluate the scenario in which the piston cylinder diameter is specified in the requirements, linked to the computation, and then the same computation parameter is linked to the graphical design. This use case allows us to evaluate chain propagation without the interference of tool operations. This process takes *0.079* milliseconds on average.

Finally, the evaluations so far only consider situations with small numbers of links. In order to evaluate how the execution of links scale, we considered the *one-to-n* use case, and timed it for different values of *n*, which is presented in Figure 5. The one-to-one propagation gives the same result as mentioned before. For n between 2 and 4, the propagation time *increased linearly*, as expected, with around *0.04* milliseconds for each additional link.

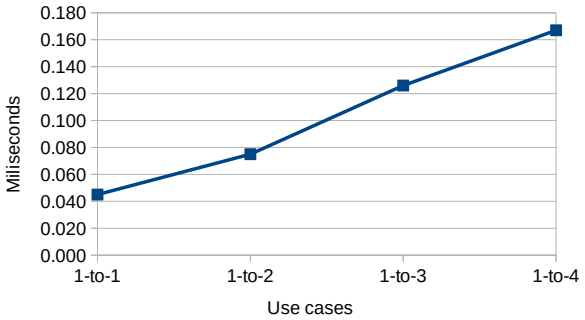


Figure 5: Execution time to propagate 1-to-*n* links, varying the value of n from 1 to 4

> ***Answering RQ2:*** *In addition to preventing human error in a wide variety of linking cases, the reactive links also greatly reduce the time required to solve the inconsistencies between related values. As opposed to other change propagation solutions, the execution time of the links is very small and the result does not require additional developer interaction.*

6 LIMITATIONS AND THREATS TO VALIDITY

In the following, we discuss the limitation of our solution and the internal and external threats related to our study.

6.1 Limitations

One limitation of our solution stems from the fact that our links cannot deal with precision differences in numeric values. It becomes an issue when one tool handles floating point values with a higher precision than another. This situation can be solved by specifying a precision level to be applied to the checks, but finding the right precision level to encompass all the cases is complex and must be defined by engineers.

6.2 Internal Threats

There are also some internal threats issues to be considered, especially regarding the evaluation method. We considered two scenarios for the evaluation, but only the modeling part of the projects. The decisions taken during the implementation of our solution might influence the results, mostly the execution time. However, we believe this is a minor threat, as the different use cases used in the evaluation of performance considered the same implementation, only varying the number of links. Regarding the result analysis, we have limited input from industry users using our solution, and therefore we could not accurately evaluate the usefulness of the solution in a distributed environment. However, we assume that there is no difference between links and other artifacts from a collaborative perspective.

6.3 External Threats

The external threat of our work concern to the generalization of the results. We cannot claim that other scenarios or systems have the same characteristics as the ones in our evaluation. However, we have identified a fair number of corner cases and complex links within our scenarios. Testing the solution on a larger system would undoubtedly uncover additional issues, as well as allow us to further evaluate the performance of the solution.

7 RELATED WORK

The synchronization of changes, especially in collaborative engineering, has been an issue that developers have encountered ever since collaboration has entered the engineering process. The problem has been addressed from multiple perspectives, each touching on certain aspects of the issue, while disregarding others. In this section, we discuss the most common solutions in the literature.

7.1 Traceability Management

The first aspect to be discussed is the maintenance of the connection between artifacts from different models or tools, that are logically related. For instance, the artifact management system ADAMS [11] uses trace links, that can be set manually or identified automatically, to mark the relationships and dependencies found between different models of the same system. The solution proposed by Rahimi et al. [35] uses a similar concept of traces, adding the possibility to adapt them to changes in the models. When a model is refactored, the solution identifies the affected trace links and suggest the necessary changes. Traces can be used to store further information of the system, which can then be part of further processing. For example, the Dapper platform [37] traces the communication within a distributed system. On the other hand, Asuncion et al. [5] suggests the usage of the Federated Information-based Design Environment to store the trace links between components, and enhance them with additional information regarding the construction and status of the system. The solution proposed by Linsbauer et al. [27] uses the traces of the dependencies between procedures and the abstract syntax tree resulted from the variant execution to recover and maintain the variability of a product line system.

Trace links are widely used in practice and have a number of advantages, but their main disadvantage from the point of view of our approach is that they establish connections at artifact level. This feature is required in many use cases and while passive, could be extended for value propagation, but the resulting coarse granularity leads to significant redundancy.

7.2 Maintaining Consistency Across Tools

Another aspect of the synchronization that is to be taken into account is the consistency of the data, both within a model, and between different related models. While the consistency within the models is usually assured by different functionalities offered by the development tools, the issue becomes harder to solve once collaboration becomes a factor. A number of possible solutions are proposed by Alvaro et al. [4], which consider multiple levels of the system: at the storage level, at runtime or at compile time through language-encoded annotations. Some solutions combine the consistency checking with traceability to cross the boundaries of models and tools. Adersberger and Philippsen [1] propose a UML extension that automatically checks for the consistency between the architecture specification and the code of the implementation. Another solution proposed by Tröls et al. [41] combines the usage of a platform for the management of artifacts across tools with traces represented in a trace matrix, and consistency rules written in Object Constraint Language (OCL [45]).

In the absence of traces, the consistency solutions limit the range of tools and languages available to the engineers. The addition of traceability features can mitigate this problem, but comes with the lack of granularity discussed previously. As a result, the rules are defined on the type of the artifacts in question. While this is a valid solution for some situations, it falls short or becomes highly inefficient when the developers are interested in the consistency of some smaller properties within some artifacts in the system.

7.3 Change Propagation

A fundamental aspect of synchronizing models is the ability to identify and solve the impact of changes across the system. This is a common problem to solve in the context of distributed and collaborative model-driven engineering. Most change propagation solutions require a lot of user involvement. This can lead to a higher risk of failure and to a slower, likely less efficient propagation. In this sense, the literature focuses on supporting the manual change propagation process, by offering suggestions on the ideal propagation paths, or predicting change propagation impact.

Determining the ideal change propagation path serves as a guide to the engineer who manually addresses the impact of the change. The solution proposed by Tang et al. [40] uses dependencies between the components of a system, stored in a matrix, to recreate all the possible propagation paths, starting from a changed component. The algorithm finds and suggest to the user the shortest and most efficient path. Alternatively, Ullah et al. [43] start by formulating a mathematical model which captures the propagation impact. This model is then used in parsing a design structure matrix, resulting in the path resulting in the best coverage of the risk. While these solutions have advantages, counteracting issues such as accidental omissions, they have the disadvantage of adding a lot of overhead to each change propagation event.

In this sense, the prediction of change propagation impact has the advantage of not firing off with every change in the system. The solution proposed by Pasqual et al. [32], for example, uses a multi-level network constructed from the history of the system. The network captures the development team organization, the product structure, and the change history, as well as the relationships between these layers. Similarly, Hamraz et al. [22] use a network containing the functional, behavioral and structural components of the system. Another solution, suggested by Ahmad et al. [2], proposes a network that traces the connection between requirements, function and component structure, and the detail design process. These matrices can then be used as a basis in change propagation algorithms and heuristics.

Change propagation solutions are useful in the practice and we have additional interested in the different component linking possibilities. However, these solutions do demand extensive of involvement of users in the correct propagation of the changes. The manual propagation is necessary in some cases, for example when complex reasoning is required, but in the majority of practical use cases automatic propagation would be both effective and efficient.

7.4 Automatic Knowledge Propagation

The relationships between the components of the system do not have to be represented as trace links. The GUI design tool Jelly [29] uses a semantic network to identify and maintain the correspondence between equivalent components in different graphical interface frameworks. When a change in a component occurs, Jelly then copies the result in the similar components of each framework. Transporting the data across tools is also possible with the solution suggested by El-Khoury et al. [14], which uses a specific framework for the development and maintenance of the data. Through this framework, the users can specify the domain description including dependencies between resources, the data allocation and the structure of the tools used. While both of these solutions address the combination of change propagation, consistency across tools and the linking of related components, their main disadvantage is that they severely limit the selection of tools and array of domains which could be used during development. Jelly does not work on any GUI development tool, and similarly, not any tool supports and implements the system suggested by El-Khoury et al. We recognize that in the industry, it is unrealistic to expect engineers to learn to use a new set of tools with every project and problem they encounter during development.

An approach that has been long available in practice consists of the usage of databases and conditions within them. Examples of such solution use consistent overlapping partitions of the data to detect, classify, handle and document inconsistencies in the system [13, 16]. These partitions, are designed to not be dependent on a certain software development method or tool set. Similarly, Grundy et al. [21] allows the specification of different kinds of views of the system, which can be linked and synchronized. Additionally, Reinert and Ritter [36] use Event-Condition-Action specifications to create links between different views of the model. These solutions have been developed over a long period of time, both in research and in the industry, and have provided valuable improvements and results. However, the main downside of these solutions is the use of a common database to store the model elements. While feasible in some cases, for example in single-domain solutions, or during the requirement specification process, many other multi-domain, multi-tool cases cannot be reduced to a single database without significant loss of information.

8 CONCLUSION

In this paper, we proposed and evaluated a solution that assures the consistency and constraint between artifacts of multi-domain and multi-tool scenarios for engineering complex system. The links proposed connect artifacts in a very fine-grained network of related property values. Then, they automatically react to changes in one of their connected values. This reaction can be customized to either propagate the change or warn the user of constraint violations introduced. The evaluation conducted showed not only the feasibility of the links, from the perspective of the use cases covered, but also that their performance can lead to great improvements in the development process. Not only are possible faults detected as soon as they are introduced, but in many cases the propagation across the links solves inconsistencies faster and more reliably than the developers could do manually.

There are, however, still some downsides to the solution, as well as development areas we have not explored. In future work, we intend to test the links on bigger systems and in real-life development environments, where engineers can give us additional feedback. Additionally, the solution presented here is only the first step, and the links can then be developed and improved to cater more closely to specific use cases. For example, our current implementation cannot handle precision differences, but an addition in this sense would vastly improve the usability of the links in more complex and thorough engineering scenarios.

ACKNOWLEDGMENT

This work has been supported by the COMET-K2 Center of the Linz Center of Mechatronics (LCM) funded by the Austrian federal government and the federal state of Upper Austria; and it was funded by the Austrian Science Fund (FWF), grand no. P31989.

REFERENCES

[1] Josef Adersberger and Michael Philippsen. 2011. ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking. In *Software Architecture*, Ivica Crnkovic, Volker Gruhn, and Matthias Book (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359.

[2] Naveed Ahmad, David C Wynn, and P John Clarkson. 2013. Change impact on a product and its redesign process: a tool for knowledge capture and reuse. *Research in Engineering Design* 24, 3 (2013), 219–244.

[3] Wasim Alsaqaf, Maya Daneva, and Roel Wieringa. 2018. Understanding Challenging Situations in Agile Quality Requirements Engineering and Their Solution strategies: Insights from a Case Study. In *IEEE 26th International Requirements Engineering Conference (RE)*. 274–285. https://doi.org/10.1109/RE.2018.00035

[4] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. 2013. Consistency without Borders. In *4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2523616.2523632

[5] Hazeline U Asuncion and Richard N Taylor. 2009. Capturing custom link semantics among heterogeneous artifacts and tools. In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. IEEE, 1–5.

[6] André Borrmann, Juha Hyvärinen, and Ernst Rank. 2009. Spatial constraints in collaborative design processes. In *International Conference on Intelligent Computing in Engineering (ICE09)* (Berlin, Germany). 1–8.

[7] Jennifer Brings, Marian Daun, Torsten Bandyszak, Vanessa Stricker, Thorsten Weyer, Elham Mirzaei, Martin Neumann, and Jan Stefan Zernickel. 2019. Model-based documentation of dynamicity constraints for collaborative cyber-physical system architectures: Findings from an industrial case study. *Journal of systems architecture* 97 (2019), 153–167.

[8] JungWon Byun, SungYul Rhew, ManSoo Hwang, Vijayan Sugumara, SooYong Park, and SooJin Park. 2014. Metrics for measuring the consistencies of requirements with objectives and constraints. *Requirements Engineering* 19, 1 (2014), 89–104.

[9] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software Traceability: Trends and Future Directions. In *Future of Software Engineering Proceedings*. Association for Computing Machinery, New York, NY, USA, 55–69. https://doi.org/10.1145/2593882.2593891

[10] Sandun Dasanayake, Sanja Aaramaa, Jouni Markkula, and Markku Oivo. 2019. Impact of requirements volatility on software architecture: How do software teams keep up with ever-changing requirements? *Journal of software: evolution and process* 31, 6 (2019), e2160.

[11] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. 2008. Adams re-trace. In *ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 839–842.

[12] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. 2015. DesignSpace: An Infrastructure for Multi-User/Multi-Tool Engineering. In *30th Annual ACM Symposium on Applied Computing*. ACM, 1486–1491. https://doi.org/10.1145/2695664.2695697

[13] Steve Easterbrook and Bashar Nuseibeh. 1996. Using viewpoints for inconsistency management. *Software Engineering Journal* 11, 1 (1996), 31–43.

[14] Jad El-Khoury, Didem Gurdur, Frederic Loiret, Martin Törngren, and Mattias Nyberg Da Zhang. 2016. Modelling support for a linked data approach to tool interoperability. *ALLDATA 2016* (2016), 51.

[15] Susan Ferreira, James Collofello, Dan Shunk, and Gerald Mackulak. 2009. Understanding the effects of requirements volatility in software engineering by using analytical modeling and software process simulation. *Journal of Systems and Software* 82, 10 (2009), 1568–1577. https://doi.org/10.1016/j.jss.2009.03.014

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

[17] Donald Firesmith. 2004. Engineering safety requirements, safety constraints, and safety-critical requirements. *Journal of Object technology* 3, 3 (2004), 27–42.

[18] Cloud Native Computing Foundation. 2022. gRPC. https://grpc.io/. Accessed: 2021-09-19.

[19] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE)*. 37–54. https://doi.org/10.1109/FOSE.2007.14

[20] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. 2018. Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map. *IEEE Transactions on Software Engineering* 44, 12 (2018), 1146–1175. https://doi.org/10.1109/TSE.2017.2755039

[21] John C Grundy, John G Hosking, and Warwick B Mugridge. 1996. Supporting Flexible Consistency Management via Discrete Change Description Propagation. *Software: Practice and Experience* 26, 9 (1996), 1053–1083. https://doi.org/10.1002/(SICI)1097-024X(199609)26:9<1053::AID-SPE51>3.0.CO;2-U

[22] Bahram Hamraz, Nicholas HM Caldwell, Tom W Ridgman, and P John Clarkson. 2015. FBS Linkage ontology and technique to support engineering change management. *Research in engineering design* 26, 1 (2015), 3–35.

[23] A.E. Hassan and R.C. Holt. 2004. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 284–293. https://doi.org/10.1109/ICSM.2004.1357812

[24] Elizabeth Hull, Kenneth Jackson, and Jeremy Dick. 2005. *Requirements Engineering in the Solution Domain*. Springer, London, 109–129. https://doi.org/10.1007/1-84628-075-3_6

[25] IBM. 2022. Advanced Rule Language (ARL). https://www.ibm.com/docs/en/dbaoc?topic=languages-advanced-rule-language-arl. Accessed: 2021-09-06.

[26] Ali K Kamrani and Emad S Abouel Nasr. 2008. *Collaborative Engineering*. Springer.

[27] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2014. Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 426–430. https://doi.org/10.1109/ICSME.2014.67

[28] Luiz Eduardo G. Martins and Tony Gorschek. 2016. Requirements engineering for safety-critical systems: A systematic literature review. *Information and Software Technology* 75 (2016), 71–89. https://doi.org/10.1016/j.infsof.2016.04.002

[29] Jan Meskens, Kris Luyten, and Karin Coninx. 2010. Jelly: A Multi-Device Design Environment for Managing Consistency across Devices. In *International Conference on Advanced Visual Interfaces*. Association for Computing Machinery, New York, NY, USA, 289–296. https://doi.org/10.1145/1842993.1843044

[30] Sarra Missaoui, Faïda Mhenni, Jean-Yves Choley, and Nga Nguyen. 2018. Verification and validation of the consistency between multi-domain system models. In *Annual IEEE International Systems Conference (SysCon)*. 1–7. https://doi.org/10.1109/SYSCON.2018.8369561

[31] N. Nurmuliani, D. Zowghi, and S. Powell. 2004. Analysis of requirements volatility during software development life cycle. In *Australian Software Engineering Conference*. 28–37. https://doi.org/10.1109/ASWEC.2004.1290455

[32] Michael C Pasqual and Olivier L de Weck. 2012. Multilayer network model for analysis and management of change propagation. *Research in Engineering Design* 23, 4 (2012), 305–328. https://doi.org/10.1007/s00163-011-0125-6

[33] Klaus Pohl. 2010. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated.

[34] Sébastien Rabeau, Philippe Dépincé, and Fouad Bennis. 2007. Collaborative optimization of complex systems: a multidisciplinary approach. *International Journal on Interactive Design and Manufacturing (IJIDeM)* 1, 4 (2007), 209–218.

[35] Mona Rahimi, William Goss, and Jane Cleland-Huang. 2016. Evolving Requirements-to-Code Trace Links across Versions of a Software System. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 99–109. https://doi.org/10.1109/ICSME.2016.57

[36] Joachim Reinert and Norbert Ritter. 1998. Applying ECA-Rules in DB-based Design Environments.. In *CAD*. 188–201.

[37] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[38] Binyang Song, NF Soria Zurita, Guanglu Zhang, Gary Stump, Corey Balon, Simon W Miller, Michael Yukish, Jonathan Cagan, and Christopher McComb. 2020. Toward hybrid teams: A platform to understand human-computer collaboration during the design of complex engineered systems. In *Design Society Conference*, Vol. 1. Cambridge University Press, 1551–1560.

[39] Antony Tang and Hans van Vliet. 2009. Modeling constraints improves software architecture design reasoning. In *Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*. 253–256. https://doi.org/10.1109/WICSA.2009.5290813

[40] Dun-Bing Tang, Lei-Lei Yin, Qi Wang, Inayat Ullah, Hai-Hua Zhu, and Sheng Leng. 2016. Workload-based change propagation analysis in engineering design. *Concurrent Engineering* 24, 1 (2016), 17–34.

[41] Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. 2019. Collaboratively Enhanced Consistency Checking in a Cloud-Based Engineering Environment. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, Article 15. https://doi.org/10.1145/3319499.3328232

[42] Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. 2021. Instant Distribution of Consistency-Relevant Change Information in a Hierarchical Multi-Developer Engineering Environment. In *36th Annual ACM Symposium on Applied Computing*. ACM, 1572–1575. https://doi.org/10.1145/3412841.3442127

[43] Inayat Ullah, Dunbing Tang, Qi Wang, and Leilei Yin. 2017. Exploring effective change propagation in a product family design. *Journal of Mechanical Design* 139, 12 (2017), 121101.

[44] Marzina Vidal, Tiago Massoni, and Franklin Ramalho. 2020. A domain-specific language for verifying software requirement constraints. *Science of Computer Programming* 197 (2020), 102509.

[45] Jos B Warmer and Anneke G Kleppe. 2003. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional.