Towards Checking Consistency-Breaking Updates between Models and Generated Artifacts

MohammadAmin Zaheri
Université de Montréal
mohammadamin.zaheri@umontreal.ca

Michalis Famelis
Université de Montréal
famelis@iro.umontreal.ca

Eugene Syriani
Université de Montréal
syriani@iro.umontreal.ca

***Abstract*—Model-based Low-Code systems rely on high-level specifications (models) to generate all artifacts of the resulting software. Such artifacts can be code, schemas, as well as data, and metadata. Maintaining consistency between models and artifacts generated from them is at the core of generative approaches in software engineering. Existing approaches have focused on the consistency problem between specific pairs of artifacts, such as models and their metamodels, class diagrams and generated code, and database schemas and data. Instead, we envision a holistic approach for maintaining the consistency that encompasses all generated artifacts. In this paper, we motivate our approach with a case study from a real model-driven software system. We identify scenarios where updates to either models or generated artifacts break consistency and outline a set of challenges and future research directions.**

***Index Terms*—Consistency Management, Low-Code Applications, Model-Driven Engineering**

I. Introduction

Low-code application development has an important role in the future of Model-Driven Engineering (MDE) [1]. Contemporary low-code development platforms are mature enough to generate entirely operational applications from high-level, domain-specific specifications, i.e., models. Many such applications are interactive and data-centric, where users continuously work with the generated system at run-time to create and store data. The generated system provides the meta-information or schema of the run-time data. During the lifespan of the software, the model, the generated meta-information, and the run-time information coexist and are subject to changes. Crucially, such changes can affect the consistency relationships between any of them.

Aspects of maintaining consistency among models, generated artifacts, and data have been studied from different perspectives, such as *Model-Metamodel Co-evolution*, *Round-trip Engineering*, *Database Schema-Data Co-evolution*, and *Model Inconsistency Management*. However, the study of these aspects has focused on specific pairs of artifacts in isolation from the others. Model-metamodel co-evolution approaches [2], [3], [4], [5] focus on the changes in the modeling languages and their instances. Round-trip engineering techniques [6], [7], [8] address inconsistency across the development cycle of a software product, ignoring data created during run-time. Database schema-data co-evolution approaches [9], [10], [11] do not take into account the consistency between models and generated artifacts. Model inconsistency management techniques [12], [13], [14] only apply to parts of the artifacts in our target low-code platforms.

However, the reality of development and run-time operation of low-code applications means that these inconsistency problems and others can appear simultaneously or in combination with one another. Thus, we are faced with the more general problem of detecting, articulating, tolerating, propagating, and resolving the inconsistencies that stem from the co-evolution of all kinds of artifacts. Here, we call this the "holistic consistency problem". It can happen at the time of modeling and specification, at run-time during operation, or during the maintenance and evolution of the system. It can also happen at different levels of abstraction and can take the form of vertical or horizontal inconsistency [15].

One of the main benefits of the MDE paradigm is that it is conducive to the development of large-scale systems [16] by allowing the management of the large and heterogeneous set of artifacts, at different levels of abstraction necessary for their development and maintenance. The MDE paradigm allows us to approach the holistic consistency problem in a unified and systematic way.

In this paper, we present a case study of inconsistencies arising in a real low-code environment. We take an MDE perspective, representing all elements of the low-code system as models and relationships between them. Using this case study, we explore the problem domain of holistic consistency and put forward a set of challenges to guide further research. We do not claim that our case study is representative of all low-code systems. Regardless, it helps highlight some of the challenges for a particular class of low-code systems where Platform-Specific artifacts are generated from a a higher-level Platform-Independent Model. We make the following contributions: 1) We identify inconsistency scenarios that can arise in the operation and evolution of a real low-code system. 2) We present a research prototype developed with Alloy [17], which showcases the functionalities present in a holistic consistency management tool. 3) We outline a set of research challenges for holistic consistency management of evolving data-centric low-code software systems.

The rest of this paper is organized as follows. We discuss related work in Section II and give background on low-code development and MDE in Section III. We present in-depth the illustrative case study in Section IV. In Section V, we outline our Alloy prototype. We discuss the set of research challenges

in Section VI and conclude the paper in Section VII.

II. Related Work

Inconsistency management in software systems is a well-studied topic. Generally, researchers have focused on the consistency of specific pairs of artifacts. We briefly overview of research in four key focus areas: *Round-trip Engineering*, *Model-Metamodel Co-evolution*, *Database Schema-Data Co-evolution*, and *Model Inconsistency Management*.

Round-trip Engineering is a development style where (a) low-level artifacts (e.g., code) are generated from high-level (e.g., models), and (b) changes in generated artifacts must be back-ported to the models [6]. Several researchers have investigated the problem of resolving the inconsistency between the low and high-level artifacts [6], [7], [8]. This is typically seen as a synchronization problem across different stages in the evolution of the system. Yu et al. [18] have developed an invariant traceability framework to resolve the inconsistencies between the user code and the template-generated code bidirectionally as artifacts evolve. Riedl-Ehrenleitner et al. [19] provide an approach to identify the inconsistencies between UML models and code. Pham et al. [20] proposed a model synchronization approach to maintain the consistency between the architectural models and generated code structure. These techniques do not address consistency-breaking changes that happen during the operation (run-time) of the system, which could impact data and artifacts at any level of abstraction. Crucially, a regeneration of the low-level artifacts making up the system is often not a viable option, as it may lead to data loss or loss of the system state.

Avoiding data loss is a major concern in research that focuses on the co-evolution of models and metamodels. This is a problem of migrating models as the corresponding modeling language evolves and is a critical concern in research in domain-specific modeling languages [21]. There is extensive research on the problem, with various approaches proposed [2], [3], [4], [5], [22], which use a great variety of techniques, such as using model transformations [3], genetic algorithms [5], and others. Others have focused on the co-evolution of metamodels and code [23]. Similar to the problem of round-trip engineering, inconsistencies caused by metamodel evolution also take place and are resolved during development and rarely during system operation. Model and metamodel co-evolution is a crucial relationship where the ("vertical", i.e., different meta-levels) consistency must be maintained. It must also be contextualized within other ("horizontal", i.e., same meta-level) consistency relationships in the generated system. We note that combinations of vertical and horizontal consistency relationships might be additionally subject to semantic, property-based constraints, in addition to the syntactic relations between models and their metamodels.

We are particularly interested in consistency relationships that involve data created and stored during the operation of low-code applications. This is a problem closely related to research in databases, where the consistency between schema and data is a classic problem. It can take various forms, such as schema repair [9], data cleaning [10], schema evolution [11], and others. Data plays a crucial role in low-code applications. Thus, holistic consistency management must also adapt and integrate techniques developed for databases.

Most existing Model Inconsistency Management approaches are applicable to parts of our target architecture. Such approaches target detecting and tracking inconsistencies [12], repairing them [13], [14], etc. While individual techniques can be applied independently to resolve specific problems, we propose creating a common, holistic approach that allows combining and coordinating them. We thus aim to identify and bridge the gaps in future work.

Overall, we note that further research is needed to combine techniques that focus on maintaining consistency during the development and evolution of systems with techniques that center the creation and storage of data during run-time operation.

III. Background: Low-Code Systems

There is currently no universally accepted definition of what constitutes "low-code" development. Based on Cabot's informal definition [24], any software engineering process where models have a fundamental role and drive the engineering tasks may be considered as Model-Driven Engineering. Low-code can then be thought of as MDE applied to forward engineering of software systems, i.e., a style of MDE for software development. More specifically, low-code may be seen as a more focused and limited version of MDE that is typically applied to the development of a specific sort of software application: data-intensive web/mobile applications.

In low-code development, the majority of application code is generated. However, developers may need to tweak and complete the generated code. There is typically a high-level platform-independent specification (in MDE terms, a model) from which the code and other relevant artifacts of a user-defined application are generated. In the next section, we present an example low-code application and explain its different components, as well as the process associated with generating.

IV. Illustrative Case Study

We present a case study illustrating evolution issues that can arise in a real low-code platform in operation. Although it is not representative of all low-code settings, yet in Section V, we generalize the lessons learned here to broader research questions.

A. Low-code platform

Alice is a researcher interested in the cocoa levels in chocolate. To explore the literature about chocolate production, she conducts a systematic review. A systematic review is a technique for searching for evidence in the scientific literature that is carried out formally, according to a well-defined process and a previously elaborated protocol [25]. Conducting a systematic review entails many steps spread out over a long period and is frequently laborious and repetitive.

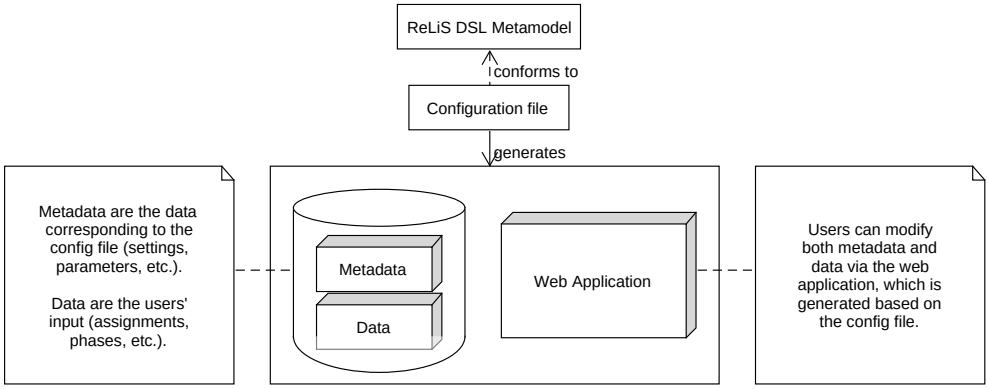

Fig. 1. Main components of ReLiS

Therefore, Alice decides to use ReLiS [26] for her project. ReLiS provides essential software support to reviewers in conducting systematic reviews. It is a model-driven low-code platform where reviewers can create a project from a protocol specification that generates a database and a web application dedicated to the project integrated into the platform.

Figure 1 depicts the main components of a system generated by ReLiS. It provides users with a domain-specific language (DSL) to define a protocol specifying how to plan, conduct, and report their review. To this end, users define a configuration file specific to their project, where they can specify, for example, the screening phases (to build the final corpus of papers), the number of reviewers and their roles, and the data extraction form (to collect relevant information from each paper). The low-code platform takes as input the configuration model and generates various artifacts to make the systematic review project operational. Thanks to the generated application, users can assign reviewers, fill forms, and conduct their systematic review.

Like all other systematic reviews, Alice's work must define a protocol and follow it while conducting her review. However, defining a protocol and getting it right in the first place is very unlikely. Therefore, she needs to improve her protocol through multiple iterations. Concretely, she will need to update the configuration file (c.f. Figure 1), which is the protocol of the systematic review in ReLiS. However, modifying the configuration file after Alice has already started conducting her review may lead to inconsistencies between the application logic and entered data of the previous iteration and the new specification.

B. Inconsistencies

We now explore a detailed example showing how inconsistencies can arise in this low-code platform while the application is in operation.

Listing 1 shows part of Alice's configuration file where she defines the data extraction form. The first line defines the name and identifier of the project. The `CLASSIFICATION` keyword instructs ReLiS to enable data extraction in this project. Then,

```
1 PROJECT chocohalice "chocohalice"
2 CLASSIFICATION
3 DynamicList cocoa "Cocoa level" [1] =
4   ["Bitter", "Bittersweet", "SemiSweet", "
        MilkChocolate"]
```

Listing 1. Systematic review project definition using the DSL of ReLiS

Alice defines the different categories and the specifications of how the data extraction form will be generated for each paper to review. ReLiS supports different types of categories. In this case, Alice defines a `DynamicList` which produces a list of values from which Alice can choose for each paper. This list is *dynamic*, meaning that she can update it while conducting the review. On line 3, this category has an identifier and a label to be displayed, and at most one value can be selected (denoted by the string "`[1]`"). The list of strings defines the initial values that will be pre-populated in the list. Figure 3 shows the generated form from where Alice can classify the papers (assign one cocoa level to each paper).

From this initial configuration file, ReLiS generates a web application in PHP encapsulating the logic of the project that runs within ReLiS itself. The dynamic architecture of this low-code platform consists of a generic application that is parameterized by the generated project-specific code [26]. Additionally, ReLiS generates a dedicated MySQL database for the project that stores metadata and user data. For the configuration in Listing 1, the system generates a `ReferenceCategory` table in the database for each dynamic list encountered, populated by the initial list of values as shown in Figure 2. This is part of the metadata. When Alice proceeds with classifying the papers, the assignment of cocoa level to each paper is stored in the database as data in a `Classification` table. This table references, via foreign keys, the cocoa level assigned to the paper, as well as other information (e.g., authors, title, etc.) not shown here for brevity. Alice can freely modify both the data and the metadata, thanks to the fully functional web application that ReLiS automatically creates and installs on the fly alongside the generated database.

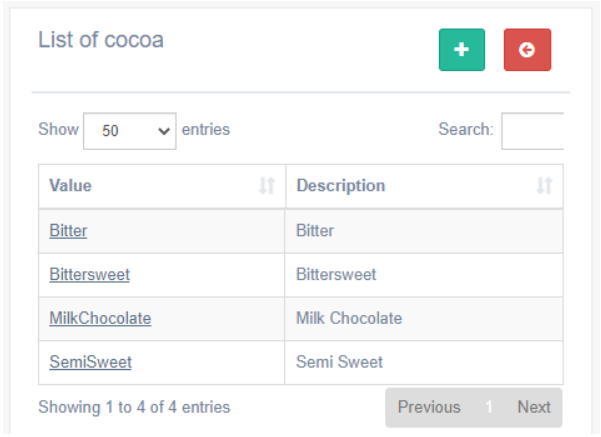

Fig. 2. Reference categories - List of cocoa levels

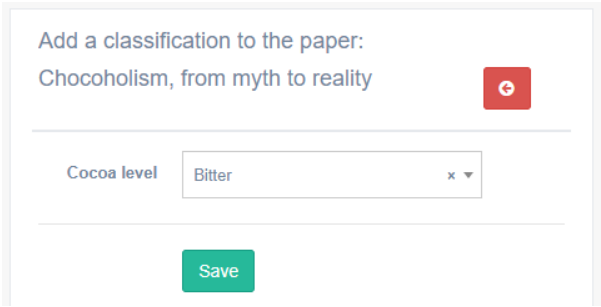

Fig. 3. The generated classification form

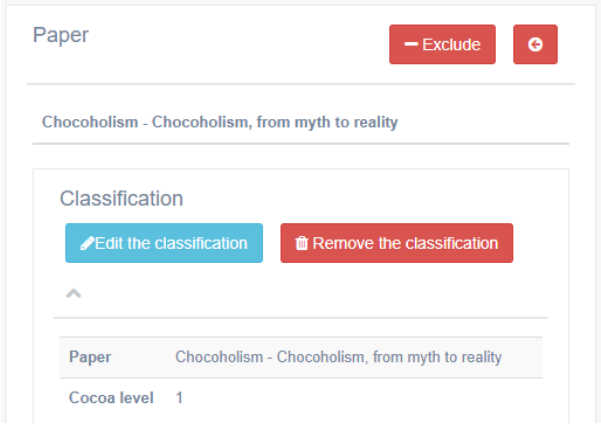

Fig. 4. A classified paper (inconsistent)

After having classified a few papers, Alice finds out that *"Bitter"* is not an appropriate value in the cocoa level category. Even though she has already assigned this value in the classification, she decides to remove it from the list of cocoa levels. She can remove it either from the running application on the `ReferenceCategory` table or from the configuration file. She first chooses the former method. The behavior of ReLiS is that, since there are already papers assigned with this value, the system automatically changes the value to 1, corresponding to the index of the item. In the web application, deleting elements only deactivates them to enable data recovery. This causes the inconsistency *(Inc1)*, shown in Figure 4. The (specification) model and the generated application are inconsistent because there is data in the model that has no corresponding translation in the application. Unsatisfied with this inconsistent state, Alice rolls back her project to the previous state (this can be done via the web application). She then removes the value following the second method: she removes *"Bitter"* from the configuration file and regenerates the application. However, she finds out there is still an inconsistency *(Inc2)* because the papers are still assigned "Bitter" while there is no corresponding value in the configuration file.

These two examples of inconsistency show the *accidental complexity* imposed by the low-code platform. To reduce this complexity, Alice would like to have a way to *(O1)* detect whether a specific change operation is breaking consistency. Also, she wants to know *(O2)* what inconsistencies will arise in different components of her project after a modification. For instance, she wants to know the implications of her decision to remove one cocoa level via the application, while some papers are classified as "Bitter", and the value "Bitter" still exists in her configuration file. If breaking consistency is unavoidable, Alice would like to know *(O3)* the least amount of inconsistency that the application must tolerate to continue her work. Finally, as she might make a set of complex changes in her project and encounter inconsistency at some point, she would like to *(O4)* be assisted to find a specific operation (or a set of specific operations) that would allow her to achieve consistency (or to see whether such operations could be found). We discuss these four objectives *(O1)–(O4)* and their characteristics more precisely in Section VI.

It is worth noting, the developers of ReLiS have implemented some consistency enforcement mechanisms. Such rules exist in many software systems; however, they are typically hard-coded and not flexible or easily changeable by the user. We are interested in the more general question of allowing different kinds of inconsistencies of different severities based on an explicitly defined consistency policy. In our example, we do not assume any given consistency policy, which allows us to demonstrate the different inconsistencies. When Alice removes the value using the second method and regenerates, the system removes all existing classifications, thus forcing her to reclassify the papers: i.e., there is data loss. This is an example of a hard-coded policy in ReLiS that causes avoiding inconsistency by removing the old classifications and generating a new version of the application. However, we should consider that some less severe inconsistencies may be allowed to emerge by design. Systems may tolerate less severe inconsistencies and let the users proceed with their tasks, as long as they can perform what they desire. For example, Alice may add new values to the `ReferenceCategory` in the web application. This still breaks consistency with the configuration file. However, she might still be satisfied with her protocol until the end of her project. By not regenerating the system, the inconsistency will not result in data loss. Therefore, the severity level of the inconsistencies may vary in each scenario.

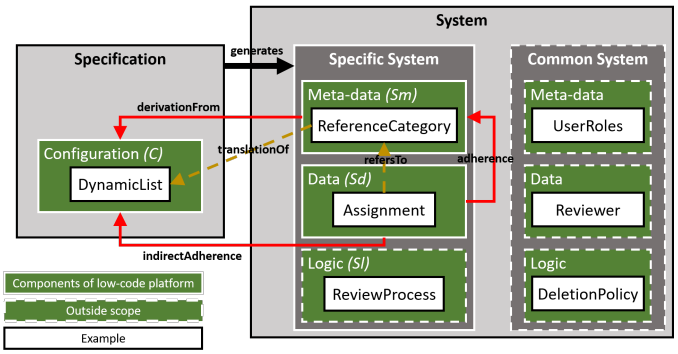

Fig. 5. Components of the considered low-code platforms and their relations

C. Consistency relations in ReLiS

To study inconsistency in a low-code system, we first need to describe the different relations between its components. Figure 5 shows a relevant slice of the artifacts involved in the process of generating a ReLiS instance. The scope of low-code platforms we target starts with a user-defined specification from which a software system is generated. Typically, not all the software of the system is generated, so we distinguish between the parts of the system common to any application generated by the platform and those specific to the specification. The specification typically consists of a configuration C like the ReLiS configuration file. The generated system typically consists of metadata Sm that stores and encodes information directly coming from C, like the classification schema and the definition of the `ReferenceCategory` table. Sm also includes system-generated data, such as the list of cocoa levels. The user-generated data Sd stored in the application conforms to Sm. For example, this can be the assignment of values from classification categories to each paper. Other aspects of the application are generated to enforce a logic of computation, and different user processes Sl, such as the review process to follow (e.g., a screening phase, followed by data extraction). Based on these, we can outline the model-level relations between different components found in these kinds of low-code platforms: (a) a *derivation* relation between the metadata Sm and the configuration C; (b) an *adherence* relation between the data Sd and the metadata Sm; (c) an *indirect adherence* relation between the data Sd and the configuration C. These relations can be further defined in terms of the specific elements of the different components. They can also be assigned additional meaning based on a given application. Below, we describe how these relations are defined in ReLiS.

We start by defining the following element-level relations in ReLiS:

- $refersTo(Assignment, ReferenceCategory)$: It is a relation between the entries of *Assignment* and of *ReferenceCategory*, which implies that each user-generated detail is assigned to one object from the collection of system-generated metadata. In our example, $P1$ is a paper defined by Alice, and it refers to *"Bitter"*, an item in the *ReferenceCategory* list.
- $translationOf(ReferenceCategory, DynamicList)$: Based on this relation in ReLiS, every single item in *DynamicList* must have a corresponding value in the *ReferenceCategory* list. In Alice's project, each item shown in Fig. 2 has an equivalent in the model defined in Listing 1.

With these element-level relations as the basis, we can define the meaning of the model-level relations for ReLiS and similar low-code applications:

- $derivationFrom(Sm, C)$: Sm is constructed using all relevant information in C, plus additional system-specific information. This could be seen as a refinement process, so that if C satisfies a property, so must Sm. In ReLiS, the derivation relation between Sm and C holds if there is a $translationOf$ relationship between *DynamicList* and *ReferenceCategory*. In our example, there is a $derivationFrom$ relationship between the generated application and Alice's configuration file, because the $translationOf$ relation maps every item of cocoa levels in the configuration file (Listing 1), to one item of cocoa levels in the application (Figure 2).
- $adherence(Sd, Sm)$: There is a relation between Sd and Sm, such that all records to be inserted in Sd must meet the constraints imposed by Sm. In the database of Alice's work, for instance, there is a foreign key in the table that stores the assignment information of $P1$, referring to the table that contains different cocoa levels.
- $indirectAdherence(Sd, C)$: There is a syntactic transitive relation between Sd and C, such that the information in Sd should adhere to the constraints in Sm, which is derived from C. For example, if a paper in Sd is classified by a V value (V defined in Sm), then there exists a translation of V to some V' defined in C ($translationOf$ relation). In Alice's project, the `DynamicList` for cocoa levels can have at most one value as specified by the constraint "`[1]`" in line 3 of Listing 1. ReLiS translates this value to a schema constraint in the database of the generated system. In other words, Alice's configuration file C specifies the generated Sm of the ReLiS project `chocohalice`. Alice's data in Sd are thus constrained through Sm by the translation (implementation) of the definitions that Alice put in C.

Even with a toy project like `chocohalice`, we can clearly see that ReLiS is a complex system with various components constrained by non-trivial consistency relationships.

D. Consistency-breaking changes

As we have seen in Alice's scenario, change can affect any of the components of the `chocohalice` project. Such changes can, in turn, put a strain on any of the consistency relationships, potentially causing inconsistency. Using the architecture shown in Figure 5 as a reference, we describe the typical inconsistencies that can occur during the operation of a ReLiS project. By extension, these inconsistencies can affect any low-code system that shares the basic architecture of ReLiS.

1) Inconsistency between C and Sm: Because of the *derivationFrom* relation between C and Sm, most changes on either component can break the consistency between them. A change in Sm may contradict its specification in C, thus being a breaking change. This is similar to the example where Alice removed an entry from the `ReferenceCategory` leading to inconsistency *Inc1*.

Conversely, a change in C may be inconsistent with Sm. For example, consider the case where Alice removes *"Bitter"* from the initial values of the `DynamicList` leading to inconsistency *Inc2*.

In both of the above cases, the *translationOf* relation between the `ReferenceCategory` and the `DynamicList` no longer holds. However, the latter case is not necessarily a severe inconsistency, since we could consider *"Bitter"* not an initial value but one added later.

2) Inconsistency between Sm and Sd: Since Sd adheres by construction to Sm, any inconsistency between them can only be caused by a change in Sm. Modifying Sm would break its consistency with C. Therefore, $Sm - Sd$ can only be inconsistent if and only if $C - Sm$ is inconsistent.

For example, Alice may decide to add a new cocoa level via the application (using the interface shown in Figure 2). Then, she can assign that cocoa level to a paper. So far, only C and Sm are inconsistent. If she removes this entry via the application, the consistency $C - Sm$ is restored. However, $Sm - Sd$ are now inconsistent because Sd no longer adheres to Sm.

Note that the consistency between C and the generated logic component (see Figure 5) cannot be broken during operation. Modifying the logic requires reprogramming the software, which is beyond the scope of changes the user of the system can perform and is thus outside the scope of this paper. Modifying C and regenerating will override the previous logic.

3) Inconsistency between C and Sd: Breaking the indirect adherence between C and Sd can also be a source of inconsistency. In our example, if Alice decides to rename one of the cocoa levels, she can either modify the *ReferenceCategory* via the application (using the interface shown in Figure 2) or the *DynamicList* within the configuration file. If she decides to change the configuration file, the inconsistency will not emerge because regenerating the system fixes the mappings. Therefore, due to the hard-coded consistency management in ReLiS, it is impossible to break the *syntactic* consistency between C and Sd by modifying C. However, if she changes the name of one cocoa level in *ReferenceCategory*, she breaks the consistency between C and Sd. To be more specific, although Sd still adheres to Sm after the change, the *derivationFrom* relationship between Sm and C does not hold anymore because such a modification breaks the *translationOf* relation. Alice can thus make C and Sd inconsistent by making a change in Sm.

The example above describes *syntactic* inconsistency. Interestingly, there is also a *semantic* consistency relation between C and Sd. This should ideally be mediated syntactically via Sm. However, it is possible in practice to bypass this by "abusing" the system. For example, suppose that Alice uses

```
1  PROJECT chocohalice "chocohalice"
2  CLASSIFICATION
3  Simple madeIn "Made in" [1] : string(32)
4  DynamicList cocoa "Cocoa level" [1] = ["Bitter"
       , "Bittersweet", "SemiSweet", "
       MilkChocolate"]
```

Listing 2. Systematic review project definition including a `Simple` field

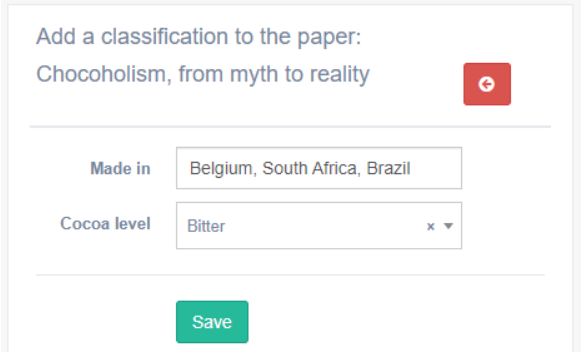

Fig. 6. The new classification form, generated based on Listing 2

Inconsistency	Type	Direction
C - Sm	Syntactic/Semantic	Horizontal
Sm - Sd	Syntactic	Vertical
C - Sd	Semantic	Vertical

TABLE I
TYPES OF INCONSISTENCIES

the revised configuration file shown in Listing 2. In line 3, this new configuration file also defines the new category `madeIn` to store information about the country of origin of the chocolate. This category is defined as a simple, single value. Table I shows the type and direction of the mentioned inconsistencies.

The generated form in ReLiS is shown in Figure 6, where the `madeIn` category is represented as a free-form entry box. In regular use, Alice would be supposed to enter a single string to store the country of origin of a particular chocolate. During the course of her work, however, Alice might find this to be too restrictive. She may find that a particular paper requires her to store multiple countries of origin for a particular chocolate. It is temptingly expedient for Alice to use the free-form entry box to store this information in the same place. Changing the configuration file and regenerating `chocohalice` might be laborious and might risk her losing her previous work. So, instead of following the procedure pre-envisaged by the creators of ReLiS, Alice might decide to create an ad-hoc convention, whereby multiple countries of origin are stored in the same field as comma-separated values. Although Sd and C correctly adhere syntactically (a comma-separated string is still a single string), they are inconsistent semantically, as the original intention of the specification in C has been eroded.

This is an example of a phenomenon that we could call "strategic misuse", which is often necessary from the user's point of view. In some cases, following the "proper" procedure is cumbersome and error-prone, and instead, the user sees it worth taking an expedient shortcut. However, this might also

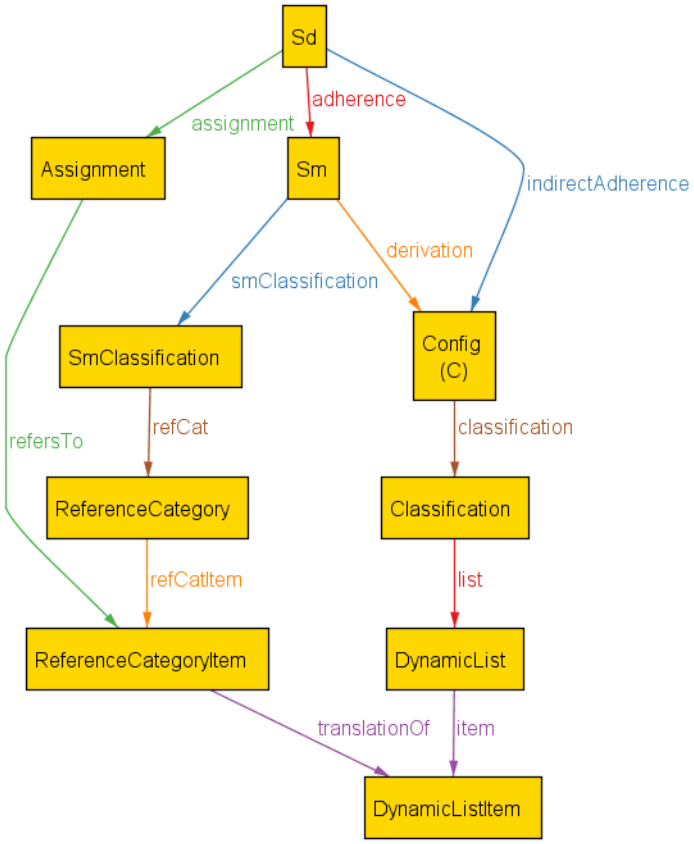

Fig. 7. The metamodel of the prototype (extracted by Alloy)

```
1  sig State {
2    config: one Config,
3    sm: one Sm,
4    sd: one Sd,

6    classification: getConfig[this] ->
7      one Classification,
8    list: getClassification[this] ->
9      one DynamicList,
10   item: getDynamicList[this] ->
11     some DynamicListItem,

13   smClassification: getSm[this] ->
14     one SmClassification,
15   refCat: getSmClassification[this] ->
16     one ReferenceCategory,
17   refCatItem: getReferenceCategory[this] ->
18     some ReferenceCategoryItem,

20   assignment: getSd[this] ->
21     some Assignment,

23   refersTo: getAssignment[this] ->
24     getReferenceCategoryItem[this],
25   translationOf: getReferenceCategoryItem[this] ->
26     getDynamicListItem[this],

28   derivation: getSm[this] one->
29     getConfig[this],
30   adherence: getSd[this] one->
31     getSm[this],
32   indirectAdherence: getSd[this] one->
33     getConfig[this]
34   }
```

Listing 3. State signature used in formalization

inadvertently cause unforeseen negative consequences. For example, some automated analysis might no longer be able to provide useful feedback (e.g., "what percentage of chocolates are made in North America?"). Detecting such strategic misuse could be an invaluable source of information for planning the future development of the software system.

V. Application: Towards Automated Management

We show a proof-of-concept prototype system that would allow us to tackle the inconsistency problems described in the previous section. We use Alloy [17], an open-source language, toolkit, and analyzer for automatic semantic analysis to articulate and formalize Alice's problem. In Alloy, a modeler creates a specification composed of a set of *signatures* and *facts*. Properties of interest are then defined as predicates over the specification. Alloy allows checking the properties by performing bounded scope model finding.

The code of our Alloy prototype is not generated; it is implemented manually. We show the relevant ReLiS metamodel slice encoded in Alloy in Figure 7. In this metamodel, we define signatures for all the concepts shown in Figure 5. We also define four additional concepts named `Classification`, `SmClassification`, `DynamicListItem`, and `ReferenceCategoryItem` to model the internal hierarchy of components implemented in ReLiS.

We formulate Alice's scenario as transitions over a finite ordered set of states. Each state represents ReLiS at a different stage in her scenario: i.e., it represents a different version of the universe that contains the entities (e.g., *Config*, *Sm*, *Sd*, *Assignment*). Each transition represents a change that Alice can make. We use the built-in Alloy module *util/ordering* to force an ordering on the states. We define a state as a signature containing the entities that represent our modeled environment. Sd is indirectly (via Sm) dependent on *parts* of C. We specify everything in Alloy and characterize the relationship between Sd and C. In the following, we describe this encoding and use it to illustrate breaking changes, non-breaking changes, and conflicts.

Listing 3 shows the `State` signature. On lines 1–4, each state contains one C, one Sm, and one Sd. Lines 6–14, define the relationships between these concepts. Finally, on lines 16–21, we show the consistency relationships that should hold as defined in Section IV-C. Listing 4 shows a sample constraint in Alice's scenario. Using the state signature, we can define different scenarios such that in the first state, the model is consistent (all constraints are satisfied), while in the next state, the model is inconsistent (some constraints do not hold).

For this purpose, we have also defined another module called *Consistency Constraints*, which contains the ReLiS consistency constraints. In an ideal world, they will never be violated (i.e., in Alloy, they would be encoded as `fact`). Nevertheless, we model them as predicates, which allows us

```
pred eachRefCatItemToOneDynamicListItem
  [s: one State] {
    side1[s] and side2[s]
  }

pred side1[s: one State] {
  all d: getDynamicListItemSig[s]
    |one r: getReferenceCategoryItemSig[s]
    |d =r.(s.translationOf)
}

pred side2[s: one State] {
  all r: getReferenceCategoryItemSig[s]
    |one d: getDynamicListItemSig[s]
    |r =d.~(s.translationOf)
}
```

Listing 4. Sample consistency constraint: each item in the *DynamicList* must have a corresponding value in the *ReferenceCategory* list.

to include or exclude them as constraints as necessary. This is a simple method of constraint relaxation, which allows us to represent inconsistencies explicitly. In other words, in this paper, by *relaxation* we mean omitting the constraint entirely. For future work, we also envision more sophisticated methods of relaxation, such as finding a weaker constraint.

Listing 4 shows a sample consistency constraint describing that there must exist an instance of `ReferenceCategoryItem` signature for each instance of `DynamicListItem` signature. To model Alice's problem, we first enforce all pre-defined constraints. Then, we execute a change operation. After the change, if we enforce the same constraints again, Alloy does not produce any instance. This means that the change operation will not succeed, and inconsistency will not emerge. To allow a change to happen, we relax a subset of the constraints after the change operation. We encode the change operations that Alice can perform using the following *transition predicates*:

- $Sm - C$ inconsistency (a change in Sm): As implemented in Listing 5, remove one instance of `ReferenceCategoryItem` signature.
  - Remove the instance from the set of `ReferenceCategoryItems` of the second state representing the inconsistent state of the universe.
  - Remove the *refersTo* between the `Assignment` and the `ReferenceCategoryItem`, which was removed in the previous step.
- $Sm - C$ inconsistency (a change in C): As implemented in Listing 6, remove one instance of `DynamicListItem` signature from the corresponding set of the second state, which represents the inconsistent state of the universe.

To find out which consistency rules the change breaks, we perform an iterative semantic analysis. We relax (remove) the constraints until we see Alloy producing results. By relaxing the constraints and trying to regenerate, we can identify which constraints are necessary to allow a consistency-breaking change to happen.

First, we run the complete specification with all constraints included before and after the transition predicate shown in

```
pred removeReferenceCategoryItem[s, s': State] {
  one d: getAssignmentSig[s], r: d.(s.refersTo) |
  {
    getReferenceCategoryItemSig[s'] =
        getReferenceCategoryItemSig[s] - r
    s'.refersTo =(getAssignmentSig[s] - d) ->
        getReferenceCategoryItemSig[s']
  }
}
```

Listing 5. Transition predicate — Sm - C inconsistency (a change in Sm)

```
pred removeDynamicListItem[s, s': State] {
  one d: getAssignmentSig[s], ri: d.(s.refersTo),
      di: ri.(s.translationOf) |
  {
    getDynamicListItemSig[s'] =
        getDynamicListItemSig[s] - di
    s'.refersTo =s.refersTo
  }
}
```

Listing 6. Transition predicate — Sm - C inconsistency (a change in C)

Listing 5. We run Alloy with a bound of 10 for all signatures, except `State`, which we limit to 2. Alloy does not find any instances. Then we relax the constraints one at a time by removing the appropriate predicates. We find that to replicate *Inc1*, we must relax constraints shown in Listing 4 and Listing 7. Alloy is then able to find an instance in 31ms. To replicate *Inc2*, we must relax constraints shown in Listing 4 and Listing 8. It takes 67ms for Alloy to find an instance. This simple method allows us to verify exactly which consistency constraints each change violates formally. In addition, we can inspect the instances discovered by Alloy to decide how to deal with the inconsistency.

VI. Challenges

In Section IV-C and Figure 5, we described the class of low-code platforms we consider relying on a configuration C, from which they generate metadata Sm, data Sd, and computation logic Sl. We described different evolution scenarios following one such platform ReLiS. Nevertheless, this is general to many other low-code systems, such as SQLMaestro [27] and Spring Roo [28]. We are interested in the evolution that may happen while the system is in operation, where changes may occur in the specification or the generated system. This may lead to inconsistencies between C, Sm, and Sd if a change occurs in any of these components. In the following, we present interesting challenges to this problem that remain open questions for future research. We refer to the objectives in Section IV-B and list the challenges related to each of them.

A. O1: Identifying consistency contraints

Consistency constraints are generally hard-wired in the system. These constraints impose that Sd, Sm, and C adhere and derive from each other as defined in Section IV-C. Additionally, the computation logic Sl imposes constraints specific to the generated application, and the common system imposes constraints common to all generated systems in the given

```
pred allReferenceCategoryItemsInReferenceCategory[
    s: one State] {
  all r: ReferenceCategoryItem |r in
      getReferenceCategoryItemSig[s]
}

pred
    eachAssignmentRefersToOneReferenceCategoryItem
    [s: one State] {
  all d: getAssignmentSig[s] |one r:
      getReferenceCategoryItemSig[s] |d.(s.
      refersTo) =r
}
```

Listing 7. Consistency constraints to relax to replicate *Inc1*

```
pred allDynamicListItemsInDynamicList[s: one
    State] {
  all d: DynamicListItem |d in
      getDynamicListItemSig[s]
}
```

Listing 8. Consistency constraint to relax to replicate *Inc2*

low-code platform. In practice, some consistency constraints are never specified in the system but that users impose on themselves, such as naming conventions or enforcing stronger constraints than the system. It is crucial to identify all these constraints and encode them explicitly in a way that allows us to reason about them. After identifying the constraints, we need to automatically express them in a common formalism. For instance, if a platform like Alloy is used, then this means generating the Alloy specification. Some of the constraints are hard to identify and encode because they involve inferring constraints from the data, going from Sd to C. These "semantical" constraints are important to identify as they can lead to practices that interfere with the correct execution of the system.

B. O2: Detecting consistency-breaking change operations

In Section V, we presented a method to encode consistency constraints and detect when a change breaks a constraint. A challenge is to detect which change operation may break these constraints. One possibility is to verify the satisfiability of the constraints before applying a change operation. However, one should evaluate the trade-off between checking all constraints a priori and performance issues that such costly operations may incur. For example, the user may revise C and apply changes while the generated system is in operation, making Sd or Sm inconsistent with the new changes. The concept of consistency severity is crucial to define to what extent the system should tolerate breaking consistency constraints and enable more flexibility to the user [29].

C. O3: Finding the minimal consistency constraint relaxation

As we have seen in Section V, we need to identify which constraint to relax in order to check consistency breaking changes. However, this is not trivial. A challenge is to derive a general procedure to select such constraints and improve the efficiency of checking the changes. A brute-force approach would try all possible combinations of n constraints exhaustively, leading to exponential complexity $\Sigma_{i=1..n}^{n}\binom{n}{i}$. For example, if there are 5 constraints in the system, we would need to try relaxing 31 different combinations of constraints.

Therefore, another challenge is to find the fewest possible constraints that we need to relax so that a specific change can happen without breaking consistency. This can be an optimization problem where we find the least inconsistent version that lets a change happen. This could also be encoded as an instance of the *MaxSAT* [30] problem, which might be particularly appropriate for a solution in Alloy, a SAT-based reasoner. Alternatively, we can capture the fact that not all consistency constraints have the same severity or importance among users. This can be modeled as a Fuzzy Constraint Satisfaction Problem which uses concepts from fuzzy sets, fuzzy logic, and possibility theory [31] to solve constraint problems that involve preferences [32].

D. O4: Proposing change operations to restore consistency

Once a change operation break consistency, we can retain the previous consistent state and the current inconsistent state. Although the user can still operate the system while it is in an inconsistent state, she will want to restore consistency eventually. A challenge is to find the change operation(s) needed to satisfy the consistency constraints while taking into account the operations that occurred during the inconsistent state. Rolling back all the changes is not always feasible since we may lose all the changes since the last consistent state. Also, applying some changes (transition predicates discussed in Section V) in reverse order may still lead the system to an inconsistent state. Therefore, we must identify the feasibility of restoring consistency given a specific set of change operations and producing the corresponding transition predicates.

However, restoring consistency is not necessarily a mechanical procedure. This is because of the various kinds of *uncertainty* in the restoration process. Uncertainty can affect many aspects of the models [33]. For example, one source of uncertainty is the problem of having to choose between alternative inconsistency resolutions without necessarily having enough information to make the best choice [34]. Other crucial sources of uncertainty are deciding on the best time to resolve the inconsistency, deciding on the level of severity of inconsistency that we are willing to tolerate, or which consistency constraints are important. In order to keep track of the points of uncertainty, as well as the dependencies between them, we envision using a dedicated language for documenting uncertainty, such as the one proposed by Dhaouadi et al. [35].

VII. Conclusion

The development and operation of interactive, data-centric low-code systems require a holistic approach to maintaining of horizontal and vertical consistency. Thus, we must develop techniques for detecting, articulating, tolerating, propagating, and resolving inconsistencies caused by the co-evolution of models, metamodels, data, and other generated artifacts.

In this paper, we have presented a case study illustrating inconsistency in the development and usage of ReLiS, a low-code application in use for several years. We have described the different relationships between parts of ReLiS and shown how inconsistency can arise in different scenarios. We have presented an Alloy prototype implementation, where we formalized the consistency constraints and the changes and showed how they could be checked. We have also shown the importance of living with inconsistency by relaxing some consistency constraints to allow useful consistency-breaking changes to happen. By analyzing the inconsistency in Alloy, we described a set of challenges in maintaining consistency between the different artifacts and outlined future research directions.

REFERENCES

[1] A. Bucchiarone, F. Ciccozzi, L. Lambers, A. Pierantonio, M. Tichy, M. Tisi, A. Wortmann, and V. Zaytsev, "What is the future of modeling?" *IEEE Software*, vol. 38, no. 2, pp. 119–127, mar 2021.

[2] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *2008 12th International IEEE Enterprise Distributed Object Computing Conference*. IEEE, sep 2008.

[3] A. Cicchetti, D. D. Ruscio, L. Iovino, and A. Pierantonio, "Managing the evolution of data-intensive web applications by model-driven techniques," *Software & Systems Modeling*, vol. 12, no. 1, pp. 53–83, feb 2011.

[4] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages," *Science of Computer Programming*, vol. 76, no. 12, pp. 1223–1246, dec 2011.

[5] W. Kessentini, H. Sahraoui, and M. Wimmer, "Automated metamodel/model co-evolution: A search-based approach," *Information and Software Technology*, vol. 106, pp. 49–67, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301915

[6] S. Sendall and J. Küster, "Taming model round-trip engineering," in *In Proceedings of the Workshop 'Best Practices for Model-Driven Software Development*, 2004.

[7] E. V. Paesschen, W. D. Meuter, and M. D'Hondt, "SelfSync: A dynamic round-trip engineering environment," in *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2005, pp. 633–647.

[8] L. Angyal, L. Lengyel, and H. Charaf, "A synchronizing technique for syntactic model-code round-trip engineering," in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)*. IEEE, mar 2008.

[9] J. Wijsen, "Database repairing using updates," *ACM Transactions on Database Systems*, vol. 30, no. 3, pp. 722–768, sep 2005.

[10] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for capturing data inconsistencies," *ACM Transactions on Database Systems*, vol. 33, no. 2, pp. 1–48, jun 2008.

[11] D.-Y. Lin and I. Neamtiu, "Collateral evolution of applications and databases," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops - IWPSE-Evol '09*. ACM Press, 2009.

[12] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 188–204, mar 2011.

[13] D. Kolovos, R. Paige, and F. Polack, "Detecting and repairing inconsistencies across heterogeneous models," in *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, apr 2008.

[14] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards automated inconsistency handling in design models," in *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*. Springer International Publishing, 2010, pp. 348–362.

[15] K. Vanherpen, J. Denil, I. Dávid, P. De Meulenaere, P. J. Mosterman, M. Torngren, A. Qamar, and H. Vangheluwe, "Ontological reasoning for consistency in the design of cyber-physical systems," in *Workshop on Cyber-Physical Production Systems*, 2016, pp. 1–8.

[16] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 396–414, may 2017.

[17] D. Jackson, *Software Abstractions, Revised Edition Logic, Language, and Analysis*. MIT Press, 2011.

[18] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012.

[19] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, "Towards model-and-code consistency checking," in *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, jul 2014.

[20] V. C. Pham, A. Radermacher, S. Gerard, and S. Li, "Bidirectional mapping between architecture model and code for synchronization," in *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, apr 2017.

[21] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai, "Automatic domain model migration to manage metamodel evolution," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 706–711.

[22] 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, jan 2016.

[23] D. E. Khelladi, B. Combemale, M. Acher, O. Barais, and J.-M. Jézéquel, "Co-evolving code with evolving metamodels," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, jun 2020.

[24] Jordi Cabot, "Low-code vs model-driven: are they the same?" [Online; accessed 10-July-2021]. [Online]. Available: https://modeling-languages.com/low-code-vs-model-driven/

[25] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-Based Software Engineering," in *International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 273–281.

[26] B. Bigendako and E. Syriani, "Modeling a tool for conducting systematic reviews iteratively," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS - Science and Technology Publications, 2018.

[27] SQLMaestro contributors, "Sqlmaestro," [Online; accessed 10-July-2021]. [Online]. Available: https://www.sqlmaestro.com/

[28] Spring Roo contributors, "Spring roo," [Online; accessed 10-July-2021]. [Online]. Available: https://projects.spring.io/spring-roo/

[29] I. David, E. Syriani, C. Verbrugge, D. Buchs, D. Blouin, A. Cicchetti, and K. Vanherpen, "Towards inconsistency tolerance by quantification of semantic inconsistencies," in *Workshop on Collaborative Modelling in MDE*, vol. 1717. CEUR-WS.org, 2016, pp. 35–44.

[30] L. C. Min and M. Felip, "Maxsat, hard and soft constraints," *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 613–631, 2009.

[31] D. Dubois, H. Fargier, and H. Prade, "Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty," *Applied Intelligence*, vol. 6, no. 4, pp. 287–309, oct 1996.

[32] F. Rossi, K. B. Venable, and T. Walsh, "Preferences in constraint satisfaction and optimization," *AI Magazine*, vol. 29, no. 4, p. 58, dec 2008.

[33] J. Troya, N. Moreno, M. F. Bertoa, and A. Vallecillo, "Uncertainty representation in software models: a survey," *Software and Systems Modeling*, pp. 1–31, 2021.

[34] M. Famelis, R. Salay, and M. Chechik, "Partial models: Towards modeling and reasoning with uncertainty," in *Proc. of ICSE'12*. IEEE, 2012, pp. 573–583.

[35] M. Dhaouadi, K. M. Spencer, M. H. Varnum, A. M. Grubb, and M. Famelis, "Towards a generic method for articulating design-time uncertainty," *Journal of Object Technology*, 2021.

[36] G. Kanakis, D. E. Khelladi, S. Fischer, M. Tröls, and A. Egyed, "An empirical study on the impact of inconsistency feedback during model and code co-changing." *The Journal of Object Technology*, vol. 18, no. 2, p. 10:1, 2019.

[37] R. Jongeling, J. Fredriksson, F. Ciccozzi, A. Cicchetti, and J. Carlson, "Towards consistency checking between a system model and its implementation," in *Communications in Computer and Information Science*. Springer International Publishing, 2020, pp. 30–39.

[38] D. E. Khelladi, B. Combemale, M. Acher, and O. Barais, "On the power of abstraction," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, jun 2020.