Catch Me If You Can: Detecting Model-Data Inconsistencies in Low-Code Applications

MohammadAmin Zaheri*, **Michalis Famelis***, **and Eugene Syriani***
*Université de Montréal, Canada

ABSTRACT Low-Code Development Platforms (LCDPs) offer the benefit of rapid application development, but they sometimes result in inconsistencies while the generated application is in operation. Such inconsistencies often occur despite passing technical validations, indicating that the generated application functions properly without errors. However, issues arise due to semantic discrepancies, leading to conflicting stakeholder perspectives on shared data. The inconsistencies can emerge from model and data co-evolution, but existing inconsistency management techniques, e.g., in databases, multi-view and multi-paradigm modeling, are not well suited to the particular challenges in LCDPs. These approaches are inadequate in this context as they rely on relationships and adherence, such as conformance, which are not applicable in LCDPs. We present a technique and formalization for detecting inconsistencies between various artifacts based on their corresponding rules in low-code applications. We evaluate the correctness of our approach on a domain-specific low-code platform, and assess its scalability, sensitivity to rule mapping complexity, and efficiency with experiments using synthetic data. The results show that the proposed approach is capable of detecting inconsistencies while maintaining a desirable level of efficiency, scalability, and sensitivity.

KEYWORDS Consistency Management, Low-Code Applications, Model-Driven Engineering

1. Introduction

Low-Code Development Platforms (LCDPs) have become increasingly popular as they provide an accessible graphical interface for the rapid development of custom applications. They allow non-technical users ("citizen developers") to build applications without needing extensive programming knowledge (Di Ruscio et al. 2022). Creating software solutions with LCDPs involves defining high-level specifications or models, and then leveraging automation to generate and deploy applications that end-users can utilize. The generated solution encompasses various artifacts, including runtime data, metadata for configuring the application (i.e., models), and source code.

In low-code development, inconsistencies can stem from the co-evolution of models and data at runtime within the platform, or from the absence of well-defined relationships, such as conformance, between various components in LCDPs (Zaheri et al. 2021). Furthermore, within a low-code environment, citizen developers are actively involved in modifying functionalities while the generated application is operational. This aspect distinguishes low-code development from other paradigms, as citizen developers do not adhere to the same guidelines and development practices as platform developers. Consequently, enforcing built-in consistency mechanisms becomes challenging in this context (Zaheri 2022).

Existing inconsistency management techniques, such as those developed for database evolution (Lin & Neamtiu 2009), round-trip engineering (Sendall & Küster 2004; Egyed 2011), and model evolution (Khelladi et al. 2020), fall short of providing a comprehensive detection mechanism suitable for LCDPs. Most of these techniques are only relevant during the design phase, primarily focusing on identifying syntactic violations. However, they may not effectively address non-syntactic issues even when the syntax is correct. We provide further elaboration

JOT reference format:
MohammadAmin Zaheri, Michalis Famelis, and Eugene Syriani. *Catch Me If You Can: Detecting Model-Data Inconsistencies in Low-Code Applications*. Journal of Object Technology. Vol. 23, No. 1, 2024.
http://dx.doi.org/10.5381/jot.2024.23.1.a5

on this topic in Section 8. Instead of these methods, we need inconsistency management techniques specific to the particular challenges of LCDPs (elaborated on in Section 3.2). Such techniques are crucial for the long term success of low-code applications (Di Ruscio et al. 2022). Inconsistency management has also been extensively studied in model-driven engineering (Lucas et al. 2009), with the goal to repair inconsistent model views using specialized "consistency rules" (Mens & Van Der Straeten 2007).

We take the view of (Dávid 2019), where consistency is defined based on the agreement of the evaluation of properties in different views. In the context of LCDPs, these properties are evaluated by technical and business rules that are encoded in the platform and the generated application for data validation. This consistency problem pertains to conflicting perspectives of the roles involved in the development of a low-code application, with respect to existing data.

In this paper, we aim to address the problem of consistency management in data-centric LCDPs, their specifications, and their generated artifacts, where inconsistencies can happen at runtime due to model/data co-evolution. Such inconsistencies emerge in the same or different levels of abstraction and, thus, can have the form of horizontal or vertical inconsistency (Vanherpen et al. 2016). We focus on the problem of inconsistency detection in LCDPs, aiming to help administrators and application developers detecting data inconsistencies and non-compliance with business rules. Our emphasis is on identifying inconsistencies, as opposed to resolving them, so that the citizen developers and application users have the necessary information to make informed choices with regards to the generated application. The provision of feedback on the outcomes of their decisions, as well as the resolution of inconsistencies, are topics for future exploration.

In our previous work, we investigated the concept of consistency in LCDPs, illustrating scenarios where artifacts were technically correct but lacked business logic accuracy. We also outlined a set of research challenges to guide further exploration (Zaheri et al. 2021). Here, our focus shifts to addressing the research challenges of identifying and detecting inconsistencies in low-code setups, and we make the following contributions:

1. Explanation of the distinctive challenges for consistency management in LCDPs;
2. Formalization of the consistency problem for LCDPs;
3. Novel approach to detect inconsistencies in LCDP applications.

We evaluate the effectiveness of the inconsistency detection on a case study of a domain-specific low-code platform. We also evaluate the performance of our approach with respect to sensitivity, scalability, and efficiency on synthetic data.

The paper is organized as follows. First, we provide some background on LCDPs in Section 2. In Section 3, we present a detailed running example and explain a set of challenges for consistency management in LCDPs. In Section 4, we provide a formalization of consistency for LCDPs using first-order logic. This formalization ensures clarity in the detection of inconsistencies and enables the automation of the detection process. In Section 5, we present a systematic method for detecting inconsistencies based on automated formal reasoning and modeling techniques. In Section 6, we show that our approach is capable of detecting inconsistencies in low-code applications. In Section 7, we evaluate the sensitivity, scalability, and efficiency of our approach using synthetic data. In Section 8, we discuss the large body of related work on consistency detection, and put forth the argument that it does not fully address the problem for LCDPs. Finally, we conclude in Section 9.

2. Background on Low-Code Development Platforms

According to Cabot's informal definition (Cabot 2023), low-code development is a subset of Model-Driven Engineering (MDE) where models play a fundamental role and guide software engineering tasks. Specifically, low-code development is a style of MDE applied to the development of data-intensive web/mobile applications.

In line with the definition by Di Ruscio *et al.*, we define LCDPs as platforms that aim to reduce the effort required for developing and maintaining specific types of applications, typically developed by "citizen developers" who create the high-level specifications from which most artifacts are generated (Di Ruscio et al. 2022). The term *"artifacts"* is used to refer to the various assets that make up the generated application, LCDP, and any data created by end-users during its operation. This may include code, schemas, interfaces, and data. Citizen developers and system administrators may need to occasionally manipulate and complete the generated artifacts. All components, including the high-level specifications and generated artifacts, may evolve during the operation of the software application, leading to potential inconsistencies among the various artifacts. Thus, it is important to consider the issue of consistency during the operation of a low-code application.

We distinguish between general-purpose and domain-specific LCDPs. General-purpose LCDPs, like OutSystems (OutSystems 2023) and Mendix (Mendix 2023), are designed to create a wide range of applications for various use cases and industries. These platforms typically provide a visual development environment with drag-and-drop components, pre-built templates, and integrated workflows to enable rapid application development without extensive coding. In contrast, domain-specific LCDPs are tailored to a particular domain or industry. These platforms are designed with a narrower focus to create application variants with a fixed set of workflows, business rules, and presentation options. Tools like those found in SQLMaestro tool suite (SQLMaestro 2023) fall under this category as they allow generating web applications based on a given SQL database.

Sahay *et al.* devised a taxonomy of LCDPs by categorizing a set of distinct features. They analyzed eight prevalent LCDPs to identify their similarities and differences (Sahay et al. 2020). Subsequently, Gurcan *et al.* used the taxonomy and a report by Gartner (Wong et al. 2023) to choose three market-leading

LCDPs for testing and evaluating two development scenarios. The selected platforms, OutSystems, Microsoft PowerApps, and Mendix, possess various attributes such as prebuilt components, easy application building and publishing, integration of multiple services, responsive mobile and web application building, security measures, and real-time dashboards (Gurcan & Taentzer 2021).

Both models at runtime and code generation techniques are employed in creating LCDPs, each with its own set of advantages and disadvantages (Sahay et al. 2020; Khorram et al. 2020; Bloomberg 2018). In models at runtime research, high-level specifications (models) are interpreted to manage the application's execution during operation. However, modifying them, especially if the changes break compatibility, can lead to data loss. For instance, in Mendix, which uses models at runtime, modifications to models (such as altering the maximum length of a field) that cannot accommodate existing data may result in data truncation by the interpreter. On the other hand, code generation-based platforms like OutSystems proactively prevent regeneration of the application in such scenarios.

The challenges outlined in Section 3.2 are relevant for data-centric LCDPs irrespective of whether they use code generation or models at runtime. Currently, LCDPs primarily rely on conventional syntactic validation and prevention mechanisms or opt for simplistic solutions, such as adjusting data to new constraints. In our study, we aim to demonstrate that relying solely on such conventional validation mechanisms and technical rules is insufficient. There are scenarios where artifacts fail to meet the business requirements of stakeholders even though they pass these validations and are technically correct.

After evaluating the previous works, the authors chose OutSystems (OutSystems 2023) for their study due to its code generation mechanism that allows for the exploration of the co-evolution of specifications and computation logic, as well as its active forum and community. These traits facilitated the implementation of our prototype, demonstrating the proof of concept of our approach and enabling its evaluation.

We justify our study by considering a hypothetical general-purpose LCDP, highlighting that our approach and formalization are not tied to any specific LCDP. We assess the validity of our approach using a domain-specific low-code platform with real data. For performance evaluations, we focus on OutSystems since there is insufficient data available for our domain-specific LCDP. Furthermore, we can evaluate our approach with both domain-specific and general-purpose LCDPs, as elaborated in the subsequent sections.

3. Motivation

We introduce a running example that presents a typical scenario of inconsistency with LCDP-based solutions. This illustrative scenario enhances comprehension of the complexities involved and offers a tangible context for the conceptual discussions in the paper's subsequent sections. We then outline distinctive challenges encountered in managing consistency in LCDPs.

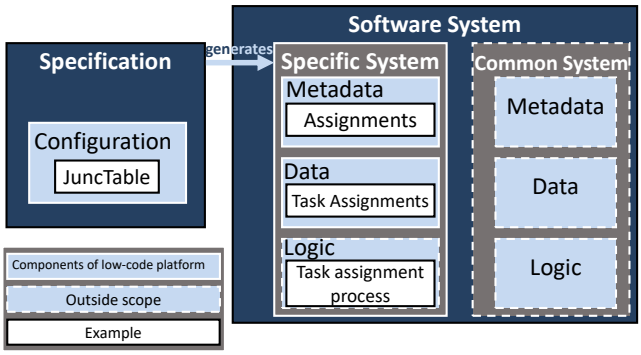

Figure 1 Components of the target low-code platforms

3.1. Running example

LCDP development involves multiple stakeholders, with different roles, goals, and agendas. To better illustrate and clarify them, as well as to motivate and explain our approach, we introduce a running example of *SimplePM*, a data-centric project management application developed using a hypothetical general-purpose LCDP, called *LoCo*. In our running example, our aim is to demonstrate scenarios where artifacts are considered technically correct, meeting all syntactic and technical requirements. However, they may not align with the intended business logic.

We emphasize that our approach and formalization is independent of any specific LCDP. The LoCo example is meant to be a stand-in for any data-centric LCDP.

Figure 1 illustrates a pertinent segment of the artifacts involved in the low-code application generation process. The low-code platform we target initiates with a user-defined specification, serving as the foundation for generating a software system. Notably, not all components of the system undergo generation. Therefore, we distinguish between elements common to any application produced by the platform and those specific to the given specification. Certain parts of the system, which include the structure of the apps generated by low-code systems, are referred to as the common system. The specific system is the part of the eventual application that is generated based on the high-level specifications in the LCDP. The white rectangles depicted in the figure represent items from our running example, aimed at enhancing clarity within the figure.

The specification typically includes a configuration, similar to the configuration panel of an LCDP. In our case, we refer to the hypothetical panel of LoCo. The configuration section includes all the modeling and specifications performed by the citizen developer (i.e., the application developer in our context) to generate the application. The generated system encompasses metadata, responsible for storing and encoding information derived directly from the configuration. This may include elements like the schema and the definition of the Assignments table, which is an instance of JuncTable. The user-generated data is stored within the application and conforms to metadata. An example of that involves task assignments to employees, that are user-generated data that comply with the constraints of the Assignments table which is an instance of JuncTable.

Moreover, other aspects of the application are generated

to enforce computational logic, accompanied by diverse user processes represented as *Logic*. These processes include, for instance, the task assignment procedure in SimplePM.

LoCo has a typical LCDP structure. LoCo involves four distinct roles depicted in Figure 2:

- The **LCDP developer** is responsible for building and maintaining the LoCo platform.
- The **application developer** is a citizen developer who uses LoCo to build and maintain SimplePM.
- The **application user** is the target user of SimplePM who performs data entry and various project management tasks. In this case, users are project managers.
- The **administrator** is another user of SimplePM with additional administrative privileges to configure SimplePM.

The first two roles operate in the Development (Dev) domain, while the other two are in the Operations (Ops) domain.

In our example, the application developer uses LoCo to build SimplePM because it provides a functionality to easily define many-to-many data relationships and imposition of data rules (noted as `JuncTable` in Figure 2). This LoCo functionality establishes a relationship between two tables with a `JuncTable`, and each record from one table can be associated with up to 1 024 records from the other table, because of technical constraints of the database. We denote this restriction as rule r_1, which is a technical rule. By technical rules, we mean guidelines and restrictions related to the technical aspects of the development process and architecture provided by the LCDP developer. The application developer, gives the administrator elevated privileges with which they can configure SimplePM by modifying some of its logic and metadata at runtime. However, the administrator cannot regenerate and redeploy SimplePM. Similarly, the application developer of SimplePM does not control the LoCo platform, which is used to develop many other applications.

To meet specific business requirements, application developers, such as the application developer of SimplePM, can instantiate and customize the `JuncTable` and its accompanying rule r_1, which are provided by LoCo. For instance, SimplePM includes a data model with tables for employees (noted as d_1 in Figure 2), tasks (noted as c_1 in Figure 2), and assignments (noted as a_1 in Figure 2). The assignments table stores the relationship between employees and tasks, and has another view or representation for the Ops domain (noted as b_1 in Figure 2). At the beginning of our illustrative scenario, the SimplePM application developer enforces two business rules in LoCo. First, a task must be assigned to exactly three employees (r_2). Second, out of the three employees assigned to a task, two must be junior and one must be senior (r_3). These rules are business rules or statements that define or constrain some aspect of a business, and adhere to LoCo's constraint for defining junction tables (r_1). The application developer also grants the administrator access privileges to modify these rules from inside the SimplePM application. This way, the administrator can adapt SimplePM to evolving business requirements without redeploying the software.

As data is an integral component of all LoCo apps, the LCDP vendor provides some default consistency management mechanisms. These mechanisms may not sufficiently address the specific concerns in the generated application and end-user content, leading to potential challenges to the overall health of the system. Inconsistencies may arise from the co-evolution of the model and data, leading to unexpected behavior and incorrect results. In our example, upper management of the company using SimplePM decided to manage resources more efficiently. Consequently, the administrator changes rule r_2 to reduce the number of employees that can be assigned to two. However, this change can result in inconsistent data, as some tasks already stored in SimplePM have more than two employees assigned to them. In other words, the data already stored in SimplePM (Assignment records) may violate the new rule (noted as r_2' Figure 2). For example, the project manager had already assigned the Triumph task to John Junior, Jane Junior, and Sam Senior. Additionally, this change causes rule r_3 to be violated, as the administrator can no longer assign exactly two juniors and one senior to a task.

We have introduced administrators to show how cross-level rule consistency might appear in LCDPs. Our objective is to illustrate the concept of cross-level consistency, where each level must adhere to the rules and constraints of the preceding level, while also possessing specific rules within their own level and domain. Although this concept is well-established in other domains, such as programming languages, it necessitates further exploration in the context of LCDPs and their resultant applications. For instance, in Python 2.x, the maximum integer value is contingent upon the execution environment, thereby imposing constraints on all integer-related rules within a Python program. In our example, we introduce the "administrator" role to demonstrate how rules at one level can interact with the constraints of the preceding level. Specifically, we assume that LoCo restricts the number of records in junction tables to a maximum of 1024 in generated applications. This technical rule is known by the app developer, who defines the business rule r_2, stipulating that each task must be assigned to 3 employees. The app developer is restricted to defining business rules that comply with the technical rules and limitations of the LCDP. In our scenario, r_2 is a specialization of r_1. Similar to the app developer, the administrator oversees assignments and has the authority to adjust business rules. Hence, the administrator must also adhere to this technical rule.

This scenario illustrates an interesting consistency management challenge. The runtime data and the application are *technically* correct in the Dev domain, as they adhere to the requirements of a correct many-to-many relationships according to rule r_1. However, they are not correct from the standpoint of the administrator, because their particular *business rules* are violated. We highlight that inconsistency stems from differing points of view regarding the existing data. More explicitly, all data in the example appear valid from the point of view of the platform's technical rules. However, when viewed through the lens of business rules, a problem emerges. The problem arises not from previously created data failing to meet business rules but from a discrepancy between the evaluation results of the technical and business rules. We identify inconsistent perspectives by examining conflicting evaluation results of technical and business rules linked with artifacts. When we refer to

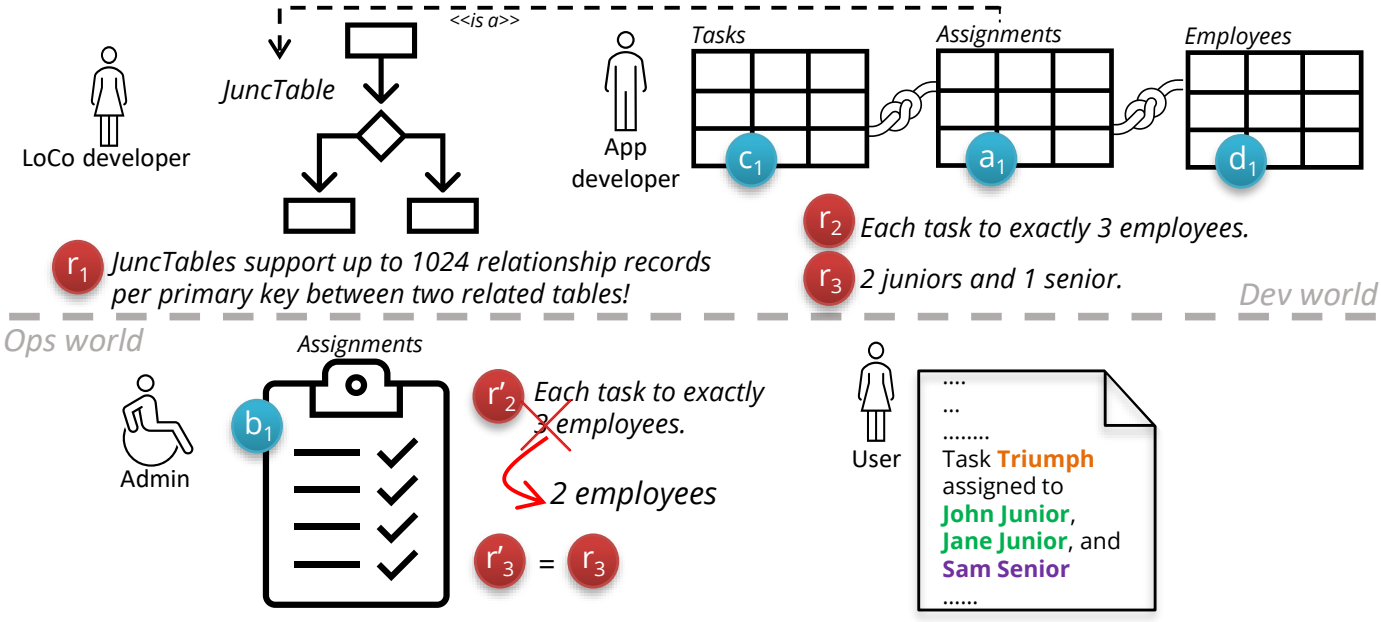

Figure 2 Roles involved in the development and operation of SimplePM, a low-code app using the LoCo LCDP

"rules", we specifically mean technical or business rules in the platform and the application, not the consistency rules used in model consistency management approaches (Reder & Egyed 2012) (Lucas et al. 2009). Such approaches define specialized consistency rules that are evaluated whenever changes occur in development artifacts. Our approach does not depend on the creation of additional, specialized consistency rules. Instead we rely on observing inconsistencies in the evaluation of rules already encoded in the LCDP and low-code applications.

Our target inconsistencies cannot be resolved by taking the system offline, as the live and operational SimplePM cannot be taken offline by the administrator, and LoCo, which may have multiple live applications, including SimplePM, cannot be taken offline by the application developer. Moreover, it may not always be desirable for the SimplePM developer to update the high-level specifications and regenerate the application, as this might cause data loss or corruption. Finally, the LCDP developer has good reasons to be conservative about changing the LoCo platform (e.g., by adding verification checks) to avoid impacting other applications running on LoCo.

Two primary approaches exist for managing inconsistencies: preventive and allow-and-recover (Dávid 2019). Preventing the emergence of inconsistencies has its own advantages and disadvantages. If LCDP developers could have prevented such issues, they would have incorporated them into their (technical) validation rules. However, research suggests that allowing inconsistencies to surface and managing, tolerating, and flagging them for user attention, rather than immediately enforcing correction or prevention, can be beneficial (Balzer 1991; Finkelstein 2000). For instance, it can help prevent premature commitment to design choices. Also, some of these inconsistencies could be workarounds that LCDP users incorporate to address actual LCDP limitations, potentially resulting in missing features. Therefore, detecting these inconsistencies and, in our future research, understanding and addressing the uncertainties before resolving them can uncover valuable insights for both the LCDP developer and the app developer and administrator.

This shows the necessity for a systematic approach to detect inconsistencies in systems where technical rules are passed, preventing system failures, while some business rules are violated.

3.2. Consistency challenges specific to LCDP

LCDPs provide various ways for defining the high-level specifications from which the eventual application is generated. These high-level specifications can be regarded as models for generating the application, similar to MDE practices. Unlike traditional modeling approaches, LCDPs store almost everything as data, presenting unique challenges in data consistency. Not all data in LCDPs are equal, warranting specialized consistency techniques. Additionally, LCDPs face distinctive pressures such as the absence of metalevel separation and the requirement for online functionality. Consequently, consistency checking in LCDPs must focus on the data-centric aspect. From our experiences in developing applications with different LCDPs such as, OutSystems, Mendix, and Microsoft PowerApps, we identify the following consistency challenges in LCDPs:

C1) **LCDP stakeholders often have conflicting points of view on what inconsistency is and how it can be managed.** For example, a change instigated by the administrator of an application causes inconsistencies from the project manager's standpoint but not from the application developer's point of view.

C2) **In contrast to MDE, LCDPs do not mandate models conforming to explicitly defined languages or metamodels.** The LCDP vendor provides functionalities to the application developers. The developers, in turn, possess the capability to permit administrators to configure segments

of logic and metadata while the system is in operation. Typically, there exists no conformance relationship between the high-level specifications and the generated application. Consequently, any modifications administrators make are accommodated, whereas the high-level specifications and the generated application do not represent identical concepts.

C3) **Business rules can be stored as (meta)data and can be modified during the operation of the application.** Like in our example, the application developer may grant the administrator privileges to modify the business rules from within the generated application. Given C2, if not managed properly, this can result in inconsistencies that are hard to detect and manage.

C4) **Regenerating artifacts from updated specifications in LCDPs may result in loss of data or system state.** In our example, even if the application developer decides to update the SimplePM specifications and regenerate the application, there is no guarantee that data already entered in the system will not be lost or corrupted. If data loss were not a concern, regenerating the application might be a simple solution to inconsistencies.

We emphasize that challenges C1 and C3, although demonstrated using the administrator role in our example, extend beyond this role. The focus of these challenges lies in the cross-level consistency phenomenon discussed in Section 3.1. Here, a modification at one level leads to cross-level inconsistency with respect to existing data, indicating conflicting perspectives among roles from different levels. Furthermore, as described in challenge C3, rule specifications in LCDPs are often stored as metadata, which can result in scenarios where during evolution, certain data components deviate from their initial specifications. Consequently, within the low-code environment, co-evolution of specifications and data can occur while the generated application remains operational.

In addition to these four challenges, we highlight the importance of *efficiency* as a requirement. We aim to develop an approach that introduces little overhead to existing LCDP consistency management mechanisms, while ensuring accurate and reliable inconsistency detection. In a low-code development setup, administrators may frequently modify rules over the application's lifespan. These changes usually happen in real-time: an administrator may edit and save a rule, potentially introducing inconsistencies, while application users continue using the system, and without observing any obvious application failure. If an inconsistency detection layer is integrated into the LCDP, it must be able to provide stakeholders accurate feedback, while avoiding the introduction of resource consumption overhead that might cause a degradation of the quality of service.

In this paper, we present an approach to efficiently tackle the four challenges by identifying inconsistencies that could emerge due to modifications in the application specification, metadata, or data.

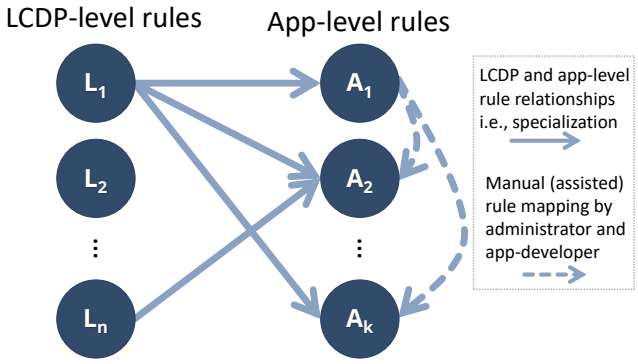

Figure 3 LCDP-level and app-level rules

4. Formalization of Consistency

We now formalize system-level inconsistency in LCDPs where a runtime application is generated from some high-level specification. Artifacts, as introduced in Section 2, are typically associated with various rules, formal or informal, explicit or implicit, that dictate how artifacts must be structured or related to one another. The violation of these rules may result in inconsistency within the system.

To capture consistency relations across different artifacts and rules, we extend the consistency definition proposed in (Dávid 2019) to encompass systems of rules. In the example of Figure 2, a_1 and b_1 are inconsistent with respect to existing assignment records, because evaluations of r_1 and r_2' on a_1 and b_1, respectively do not agree. The Triumph task assigned to three employees satisfies the technical rule r_1 *"JuncTable records per primary key* $\leq$ *1024"*, but it does not comply with the new business rule r_2' *"each task assigned to 2 employees"*. Additionally, evaluations of the two business rules r_2' *"each task assigned to 2 employees"* and r_3' *"2 juniors and 1 senior"* on b_1 do not agree, meaning that these rules are also no longer consistent with respect to existing assignment records. In other words, the system is in an inconsistent state, where the application continues to function without failure due to adherence to technical database constraints. However, this situation is risky as the inconsistency may go unnoticed and unmanaged, potentially compromising the quality of the generated application.

We highlight that when we assess a single rule on an artifact, we validate its correctness, not its consistency. However, shifting from correctness to consistency as explained in Section 3.1, and allowing inconsistencies to emerge rather than preventing them might yield valuable insights. Furthermore, analyzing pairs of rules can uncover conflicting evaluations between the Dev and Ops domains or within a single domain with respect to existing data.

We define consistency as the relationship between artifacts and their associated rules. Specifically, we consider artifacts a_x and b_y, along with their corresponding rules r_n and r_k that must be satisfied by the artifacts. We state that a_x and b_y are consistent if and only if their respective r_n and r_k are related and their evaluations on the artifacts concur. More formally, RR is the "rule traceability relationship" between r_n and r_k defined

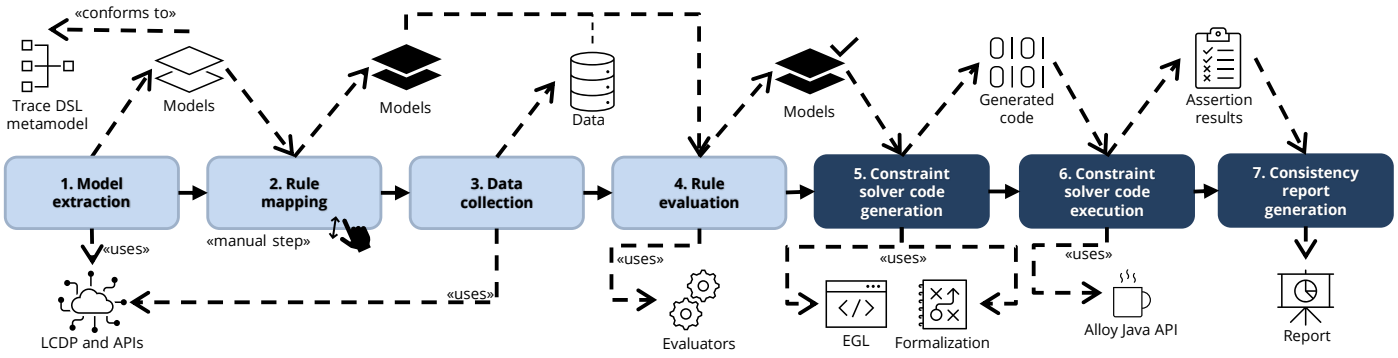

Figure 4 Overview of the approach

as:

$$RR(r_n, r_k) = RM(r_n, r_k) \wedge RE(r_n, r_k) \quad (1)$$

The predicate RM denotes a traceability mapping between the rules. In the SimplePM app example, RM captures the fact that the system generates the `Assignment` table from the specifications based on `JuncTable`, and their associated rules r_1 and r_2' are related. Moreover, the business rules r_2' and r_3' are related to each other. This relationship applies to r_2 and r_3 as well. The predicate RE captures the fact that the evaluation of the two rules against their respective artifacts should agree. In our example, the rule r_2' (*"each task two employees"*) is a specialization of rule r_1 (*"up to 1024 relationship records per primary key"*). Thus, the predicate $RE(r_1, r_2')$ means that, if they are evaluated on their associated artifacts, i.e., a_1 and b_1, they should either both evaluate to *true* or both to *false*. The same situation applies to r_2' and r_3' concerning b_1.

Figure 3 illustrates LCDP-level and app-level rules. In our example, the technical rule r_1 pertains to the LCDP-level, while the rest are business rules belonging to the app-level. App developers cannot introduce app-level rules independently; there is always a connection with an LCDP-level rule, because they are bound by the rules of the platform.

For roles that exclusively operate within the Ops domain, like administrators, mappings only involve app-level rules, which are in their domain of expertise. The relationship between app-level and LCDP-level rules can be captured automatically by the LCDP or semi-automatically by stakeholders who are proficient in both business and technical rules within the Dev domain, like app developers. In Section 5, we delve further into rule mappings.

We could specialize RR to differentiate the different kinds of traceability relationships among LCDP-level and app-level rules. However, this is not needed for our analysis. We keep our formalization abstract, allowing us to define data consistency based on the evaluation results of rules, irrespective of the rules type.

As discussed in challenge C2, in Section 3.2, LCDPs often lack well-defined meta-levels and conformance relationships between them. The rule traceability relationship RR makes relationships across levels explicit.

Given RR, we define **consistency**^{r} as follows:

$$consistent^r(a_x, b_y, r_n, r_k) \equiv RR(r_n, r_k) \implies (E(a_x, r_n) \Leftrightarrow E(b_y, r_k)) \quad (2)$$

Here, $E(a_x, r_n)$ and $E(b_y, r_k)$ are *true* if and only if a_x and b_y satisfy the rules r_n and r_k, respectively.

Importantly, r_n and r_k are *artifact-level* rules, i.e., they are evaluated on individual artifacts within the low-code system. However, we define $consistent^r(a_x, b_y, r_n, r_k)$ at the *system level*. This means that it is applicable to any pairs of related artifacts (code, models, data, etc.) at any level of abstraction within the low-code system, encompassing both the platform and the generated application. The definition of $consistent^r$ is modular, allowing to outsource the evaluation of artifact-level rules to artifact-specific reasoners or procedures.

To express the requirement that artifacts a_x and b_y are consistent with each other, we can consider the set of all rules that each should satisfy, denoted as $S^r_{a_x}$ and $S^r_{b_y}$, respectively. For instance, in the SimplePM application, $S^r_{b_y}$ includes rules r_2 (*"each task assigned to 3 employees"*) and r_3 (*"each task assigned to 2 juniors and 1 senior"*). We formalize the general consistency between a pair of artifacts a_x and b_y as:

$$consistent(a_x, b_y) \equiv \forall r_n \in S^r_{a_x}, \forall r_k \in S^r_{b_y}, consistent^r(a_x, b_y, r_n, r_k) \quad (3)$$

In our example, it evaluates to *false* for artifacts a_1 and b_1, since a_1 satisfies r_1 but b_1 does not satisfy r_2'.

By formalizing the consistency problem in this manner, we can also examine a pair of related rules on the same artifact to evaluate their consistency. In our example, following the administrator's modification, there are assignment records in their view (b_1) that meet the criteria of rule r_3' but not r_2'.

Finally, we define the consistency of a system M as the conjunction of all pairwise *consistency* predicates of its artifacts:

$$consistent(M) \equiv \bigwedge_{a_x, b_y \in L} consistent(a_x, b_y) \quad (4)$$

In our example, the SimplePM app is inconsistent, because at least one of the pairwise *consistency* predicates (i.e., $consistent(a_1, b_1)$) is false.

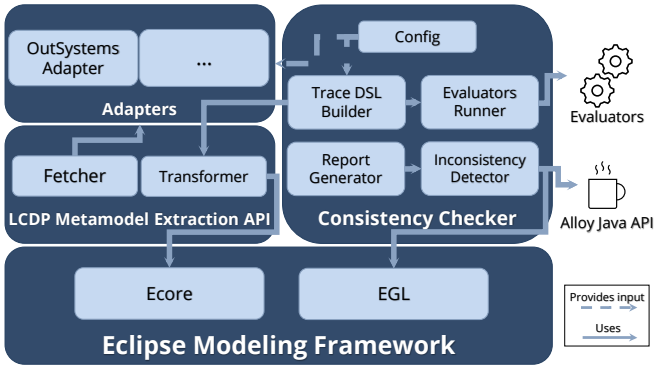

Figure 5 Architecture of the inconsistency detection framework.

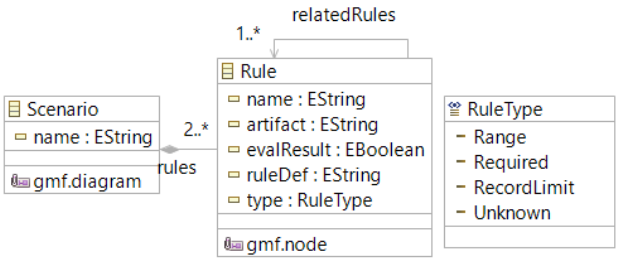

Figure 6 Rule mapping metamodel

We employ this formalization as a technique for consistency detection and implement it using Alloy, a formal tool renowned for its proficiency in rapid prototyping.

5. Inconsistency Detection

Our inconsistency detection approach is a multi-step workflow, shown in Figure 4. It consists of a preprocessing phase (steps 1–4) and the main detection phase in (steps 5–7). This workflow is completely automated using an Ant builder, with the exception of step 2: "Rule mapping", which is done manually. The inconsistency detection framework is implemented in Java using the Eclipse Modeling Framework and the Alloy Analyzer.

5.1. Preprocessing phase

The preprocessing phase (steps 1–4 in Figure 4) gathers the information required for the detection phase.

5.1.1. Model extraction The initial step creates an analyzable representation of the low-code system, including the platform, application, and user data. MDE provides a conducive framework for representing and reasoning about large-scale systems, enabling effective management of a diverse set of artifacts at various levels of abstraction (Hebig et al. 2017). Leveraging the similarities between low-code development and MDE, we can reuse existing MDE techniques and technological infrastructure. However, low-code platforms may not always use models conforming to explicitly defined metamodels, as some platforms may store data in schema-less documents or databases (Di Ruscio et al. 2022). Therefore, the first step in our approach is to reconstruct models and metamodels from the artifacts as needed, to facilitate further analysis.

We create a flexible and modular architecture that allows adapting our approach to various LCDPs, each with unique characteristics and requirements (Figure 5). While our formalization is applicable to arbitrary LCDPs, the preprocessing phase requires the LCDP to provide APIs or facilities for accessing information such as the data model or data of the generated applications.

We have defined an extraction API that can be implemented as per the requirements of a particular LCDP. In our prototype, we have specifically implemented this API for an OutSystems extractor to extract the metamodel (i.e., the data model of the tables in our running example) and the corresponding data. To accomplish this, we developed a simple low-code application as a module in OutSystems. We then created a Java interface (LCDP adapter) in our framework, and used it to extract the data and data model from our OutSystems module in CSV format. We used system entities such as *Espace*, *Entity*, *Espace_Entity*, and *Entity_Attr* for this purpose. These system entities maintain the data model of generated applications, tracking their attributes, entities, and associated constraints, which are the rules we aim to extract. The OutSystems extractor, which represents the concrete implementation of the Fetcher and the Transformer depicted in Figure 5, is responsible for converting the data obtained from the preceding entities into EMF EObjects. Acting as middleware, these extractors receive the LCDP-specific extensions and translate them into EMF EObjects. For the hypothetical LoCo platform example, we assume that this preliminary step has been previously completed. We assume that we have been able to reconstruct all artifacts in the SimplePM application as class and object models.

5.1.2. Rule mapping As described in Section 4, to evaluate the consistency between artifacts and rules, we need to know the relationships between rules RR, expressed as traceability mappings between rules. The relationships between rules can be diverse, including derivation, refinement, specialization, co-satisfaction, etc. In the case of LoCo, rule r_2' is a specialization of rule r_1, and there is a co-satisfaction relationship between r_2' and r_3'.

To capture these relationships, we create a Rule Mapping domain-specific language in Ecore, the metamodel of which is shown in Figure 6. Users, like the SimplePM administrator, can create rule mapping models that conform to this DSL manually or automatically if the LCDP provides appropriate traceability information. The stakeholders responsible for mapping the rules and establishing relationships (administrators and app developers) oversee this process within their respective domains of expertise. As depicted in Figure 3, we require creating manual mappings between app-level rules. The app developer establishes the mappings between app-level and LCDP-level rules. This role should possess expertise in both the constraints of the LCDP for app development (technical rules) and the validation rules they define to ensure their application's quality (business rules). To streamline these tasks, we envision a wizard interface

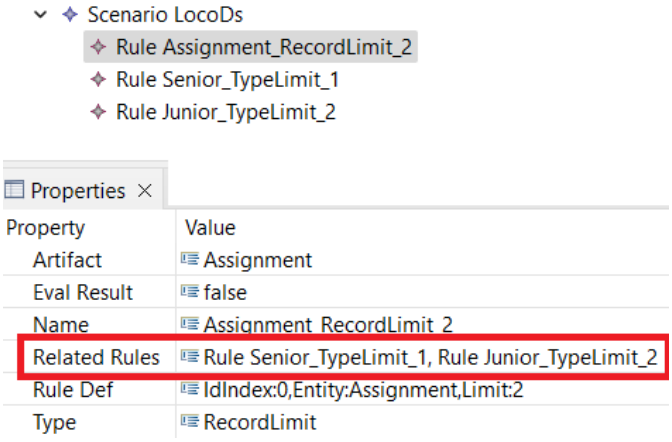

Figure 7 An example of rule mapping for SimplePM

to assist in the mapping process by allowing users to select relevant app-level rules from a pre-populated list, while handling the instantiation of the metamodel of the trace DSL behind the scenes.

We show the rule mapping of SimplePM in Figure 7. Instances of the rule mapping metamodel capture the rules, their specifications, their associated artifacts, and their relations with other rules.

In our prototype, we have implemented three different rule types. The *Range* rule type is used to evaluate whether a numerical value or the length of a string falls within a specified range. The *Required* rule type ensures that mandatory fields contain a value, and the *RecordLimit* rule type validates both the minimum and maximum allowable number of records of a particular type. In our example, both rules fall under the RecordLimit category. The *Unknown* rule type serves as a catch-all placeholder, while more rule types can be added if needed. We do not claim the universality of these three types of rules: additional rules can be included if needed. Additionally, the ruleDef attribute facilitates the creation of simulators for external rule evaluators, as detailed in Section 5.1.4.

We also link the rules to corresponding artifacts to indicate that the artifacts are expected to satisfy the rules. In Figure 7, the selected rule is linked to the `Assignment` artifact. Having this information allows us to construct RM and RE, as discussed in Section 4. The evaluation of the artifacts against the rules is handled in the rule evaluation (step 4).

5.1.3. Data collection

In this step, we collect the runtime data from the generated application and the LCDP. These data are either available in the LCDP during the generation of the application or are provided as input to the application by end-users during its operation. For example, in the SimplePM app, the rule mapping model initially represents the relationship between the two artifacts and their associated rules (a_1, r_1 and b_1, r'_2 from Figure 2). We augment it with the data fetched from the LoCo LCDP and the generated SimplePM app, i.e., the records of the `Assignment` table from Figure 2. This allows us to prepare for evaluating the related rules ($r1$ and r'_2) and (r'_2 and r'_3) on the collected data.

5.1.4. Rule evaluation

In this step, we systematically verify each artifact a_x against any rule r_n that it is associated with. This allows us to compute the predicate $E(a_x, r_n)$, which is true iff a_x satisfies r_n, as discussed in Section 4. We update the rule mapping model with the results of the external rule evaluations. In our example, we use custom external rule evaluators capable of parsing and evaluating the rule stored in the ruleDef attribute, as illustrated in Figure 7. We apply these rules to the records stored in the `Assignment` table, and store the results in the rule mapping model. As shown in Figure 7, we record the evaluation result in the `Eval Result` field of the rule mapping model.

Section 4 presents the modular definition of $consistency^r$, enabling the delegation of artifact-level rule evaluation to artifact-specific reasoners. This flexibility facilitates the integration of semi-formal or informal rules and evaluators into our framework. Our approach solely necessitates the evaluation result and is independent of the evaluator used.

In the rule evaluation stage, we do not assume any specific level of formality. The satisfaction of formally defined rules can be verified with an off-the-shelf automated reasoner, such as a constraint checker. But the same architecture can be used for rules whose evaluation is semi-formal or informal. This flexibility enables us to reason about approaches such as assurance arguments, code inspections, and more (Maksimov et al. 2020).

5.2. Main detection phase

We take the following steps (5–7 in Figure 4) to detect inconsistencies:

5.2.1. Constraint solver code generation

In Section 4, we formally defined consistency using first-order logic. We created a prototype consistency checker using Alloy (Jackson 2011), an open-source language, toolkit, and analyzer for automatic semantic analysis. In Alloy, a modeler creates a logic specification as a set of signatures and constraints over them. Properties of interest can be defined as assertion predicates over the specification. These can then be checked by bounded scope model finding, using a boolean satisfiability constraint solver (SAT solver) (Gomes et al. 2008), which makes it efficient and effective for analyzing complex systems.

We encode our formalization of consistency as an Alloy module, as shown in Listing 1. Lines 1–10, define signatures to model rules, artifacts, and their relationships in a given inconsistency checking scenario. In Lines 7–10, specify that the scenario is characterized by a set of *"givens"*, which are to be taken from the rule mapping model, created during the preprocessing phase. On lines 11–20, we define predicates defined in our consistency formalization in Section 4.

Then, we use this module to represent a concrete consistency checking scenario. A scenario is encoded in generated Alloy code, using the module defined in Listing 1 and the rule mapping model prepared during preprocessing. To generate it, we use template-based code generation with the Epsilon Generation Language (EGL) (Rose et al. 2008). We show the generated Alloy code for the LoCo example in Listing 2. Lines 2–5, define the artifact and the rules of the scenario shown in Figure 2. On lines 6–13, we define the structure of the

```
1  module Consistency
2  sig Artifact {}
3  sig Rule {}
4  sig Scenario {
5    artifacts : set Artifact,
6    rules: set Rule,
7    // These we assume given (dynamically computed
       based on the rule mapping model)
8    givenRM  : rules -> rules,
9    givenRE : rules -> rules,
10   givenE : artifacts -> rules }
11 pred RM[s: one Scenario, r1: one Rule, r2: one
       Rule] {
12   r1 -> r2 in s.givenRM }
13 pred RE[s: one Scenario, r1: one Rule, r2: one
       Rule] {
14   r1 -> r2 in s.givenRE }
15 pred RR[s: one Scenario, r1: one Rule, r2: one
       Rule] {
16   RM[s, r1, r2] and RE[s, r1, r2] }
17 pred E[s: one Scenario, a: one Artifact, r: one
       Rule] {
18   a -> r in s.givenE }
19 pred Consistency_r [s: one Scenario, a: one
       Artifact, b: one Artifact, r1: one Rule, r2:
       one Rule] {
20   RR[s, r1, r2] implies (E[s, a, r1] iff E[s, b,
       r2]) }
```

Listing 1 Formalization implemented in Alloy

scenario as a generated set of "givens" based on the rule mapping model. These "givens" encode the full interpretation of the predicates *RM*, *RE*, and *E*, defined in Section 4 for the given scenario. We highlight that `givenE` *reifies* the results of external rule evaluators. For example, if the result of applying the rule r_2 `Assignment_RecordLimit_3` on the artifact `Assignment` b_1 is `true` we add `(Assignment -> Assignment_RecordLimit_3)` to `givenE` (line 12 of Listing 2 generated for lines 17–18 of Listing 1).

5.2.2. Constraint solver code execution To verify consistency in a given scenario, we conduct assertion checking on the generated Alloy code from the preceding step. For the LoCo example, in lines 15–23 of Listing 2, we define and verify assertions $consistency_r$ for artifact `Assignment` b_1, and its associated rules r'_2 and r'_3. As we introduce one singleton signature per rule, extending the main Rule signature, we use a scope of 1 in Alloy. This results in a reduced search space and improved efficiency. We use the Alloy Java API to execute the assertion checks of the generated code.

5.2.3. Consistency report generation If an Alloy assertion is invalid, i.e., if an inconsistency is found, Alloy produces a counterexample. We take note of all invalid assertions to produce an overall consistency report for the scenario. It includes details such as the specific rule or rules that are violated, the rule mapping model elements that contribute to the inconsistencies, and any additional information that aids in understanding the root cause of the inconsistencies. In the prototype, the report is presented as a simple text document that lists all the rules and highlights those that are inconsistent with respect to the existing data. Listing 3 shows the generated consistency report

```
1  open Consistency
2  one sig Assignment extends Artifact {}
3  one sig Assignment_RecordLimit_3 extends Rule {}
4  one sig Senior_TypeLimit_1 extends Rule {}
5  one sig Junior_TypeLimit_2 extends Rule {}
6  one sig LocoDs extends Scenario {
7  } {
8    artifacts = Assignment
9    rules = Junior_TypeLimit_2 + Senior_TypeLimit_1
       + Assignment_RecordLimit_3
10   givenRM = (Assignment_RecordLimit_3 ->
       Junior_TypeLimit_2) + (
       Assignment_RecordLimit_3 ->
       Senior_TypeLimit_1) + (Senior_TypeLimit_1 ->
       Assignment_RecordLimit_3) + (
       Junior_TypeLimit_2 ->
       Assignment_RecordLimit_3)
11   givenRE = (Assignment_RecordLimit_3 ->
       Junior_TypeLimit_2) + (
       Assignment_RecordLimit_3 ->
       Senior_TypeLimit_1) + (Senior_TypeLimit_1 ->
       Assignment_RecordLimit_3) + (
       Junior_TypeLimit_2 ->
       Assignment_RecordLimit_3)
12   givenE =  (Assignment ->
       Assignment_RecordLimit_3) + (Assignment ->
       Senior_TypeLimit_1) + (Assignment ->
       Junior_TypeLimit_2)
13 }

15 assert
       Assignment_RecordLimit_3__Senior_TypeLimit_1
       {
16   Consistency_r[LocoDs, Assignment, Assignment,
       Assignment_RecordLimit_3, Senior_TypeLimit_1]
17 }
18 check
       Assignment_RecordLimit_3__Senior_TypeLimit_1
       for 1

20 assert
       Assignment_RecordLimit_3__Junior_TypeLimit_2
       {
21   Consistency_r[LocoDs, Assignment, Assignment,
       Assignment_RecordLimit_3, Junior_TypeLimit_2]
22 }
23 check
       Assignment_RecordLimit_3__Junior_TypeLimit_2
       for 1
```

Listing 2 Generated Alloy code for the LoCo example

of the running example, illustrating the inconsistency between two rules with respect to existing data. The current version of our prototype generates a basic text file listing the rules and artifacts, along with their consistency status with respect to existing data. The first line of the text report in Listing 3 shows that the running example contains inconsistencies. Lines 3–7 highlight situations where, despite adhering to the updated rule of assigning a maximum of 2 employees to a task, other rules (i.e., having 1 senior and 2 juniors) cannot be satisfied, causing the system to enter an inconsistent state. Future iterations of the prototype will furnish a well-organized response and a message that is more easily understandable to humans.

```
1 Inconsistencies found in scenario: min-
      paperProblem-easy-all-inc

3 INCONSISTENT:
4 Check min_paperProblem_easy_all_inc_Assertion__
      __Assignment_RecordLimit_2____Senior_TypeLimit_1
       for 1

6 INCONSISTENT:
7 Check min_paperProblem_easy_all_inc_Assertion__
      __Assignment_RecordLimit_2____Junior_TypeLimit_2
       for 1
```

Listing 3 Consistency report

6. Evaluation of Correctness

6.1. Objective

We conducted a case study to validate the effectiveness of our approach in detecting inconsistencies in a real domain-specific low-code platform. We pose the following research question:

RQ1: Correctness – Can our approach correctly identify inconsistencies?

6.2. Case description

ReLiS (Bigendako & Syriani 2018) is a domain-specific low-code platform geared towards supporting reviewers for conducting systematic literature reviews (SLRs) (Kitchenham & Charters 2007). With ReLiS, reviewers can create a project from a protocol specification, which generates a dedicated web application and a database integrated into the platform. We describe the user roles and usage scenarios in ReLiS using the same terminology as in the LoCo example. Project managers define a SLR protocol and generate the project: this is the app developer role of our running example, where each app is an SLR project. ReLiS supports a domain-specific language to define a protocol for planning, conducting, and reporting their review. It offers an online editor to design and modify the project-specific protocol model, where they specify various parameters, such as the screening phases, number of reviewers and their roles, and the data extraction form for collecting relevant information from each paper. Project managers can modify the protocol and redeploy an updated version of the project while the project is running. These users can make specific changes to the protocol directly in the running project via an administrator menu: this is also the administrator role of our running example. Reviewers can be assigned articles to screen and classify using the data extraction forms to effectively conduct the SLR: this is the application user role of our running example.

Changes to the ReLiS configuration specification while conducting the SLR result in the regeneration of some of these artifacts, which may produce breaking changes to the currently existing data. Furthermore, a project manager can change a subset of the configuration directly in the generated application through the administration panel without modifying the textual model. This creates inconsistencies between the rules specified in the configuration model and in the generated application. Therefore, although the SLR naturally evolves with new rules, some new rules may lead to inconsistencies because there are multiple points in which rules can be added or changed: at design-time (configuration model) and at runtime (administration panel).

Comparing ReLiS to our running example, ReLiS is an LCDP like LoCo, while each SLR project is a distinct application like SimplePM. ReLiS exemplifies one of our intended low-code platforms, as depicted in Figure 1. The configuration aspect of ReLiS is encapsulated within its configuration file, while the metadata encompasses system-generated data, such as the exclusion criteria for papers in a systematic literature review conducted within the ReLiS environment. Additionally, user-generated data can include the assignment of classification categories to individual papers within the review process.

We want to emphasize that our evaluators do not delve into the textual content of the SLR criteria; rather, they focus on detecting variations in the number of criteria. Detecting inconsistencies between the initial content and the final version of a criterion in the generated app requires a more sophisticated external rule evaluator to consider its text. However, as detailed in Section 4 and Section 5, our formalization operates independently of external evaluators and identifies inconsistencies between interconnected rules based on the evaluation results provided by these evaluators. Specifically, we compare the number of SLR criteria specified and those present in the generated app. In some SLR projects, papers may be excluded based on a criterion that is removed in the app but remains in the specification. This scenario resembles the LoCo example, where the number of Assignment records was technically correct but incorrect from a business logic standpoint.

6.3. Consistency of SLR projects

As a benchmark for validation, we conducted a study on a sample of 10 SLR projects available in ReLiS, listed in Table 1. We chose projects of a significant size that utilized a majority of the functionalities offered in ReLiS to define their protocols. Designing a protocol and getting it right on the first attempt is uncommon, and project managers often need to refine their protocols through multiple iterations. However, modifying the protocol model after starting the review process can result in inconsistencies between the application logic and the data entered in previous iterations, and the new specification.

One common aspect that may change during the course of an SLR is the exclusion criteria: reasons to exclude an article from the corpus of the review. In ReLiS, project managers can modify the exclusion criteria via the generated application. This may introduce inconsistencies between the protocol model and the generated project with respect to already existing data. Detecting and understanding these complex changes and the resulting inconsistencies can be challenging for ReLiS users. We thus can use our approach to help users identify and understand the inconsistencies.

In Table 1, we present the details of the SLR projects that we investigated. The table displays the number of exclusion criteria recorded at the outset of each project in the protocol model, as well as in the database during the project's runtime. We observe that the exclusion criteria of 5 of the projects underwent evolution over time (marked in bold). For example, one project

Table 1 SLR projects in ReLiS with the initial number of exclusion criteria in the configuration and the final number in the database

Project	Initial criteria	Final criteria	Consistent (manual)	Consistent (Alloy)
Collaborative modeling	8	8	✓	✓
Language composition	8	8	✓	✓
Language composition classification	8	8	✓	✓
Model-Driven Engineering	4	4	✓	✓
Reinforcement learning	5	5	✓	✓
Cyber physical systems	**1**	**7**	✗	✗
Dependency freshness	**3**	**4**	✗	✗
Dependency graphs	**6**	**8**	✗	✗
Mobile app development	**2**	**5**	✗	✗
Modeling assistance	**7**	**1**	✗	✗

```
1 PROJECT mobileApps "Mobile app development"
2 SCREENING
3 Reviews 2
4 Conflict on Decision resolved_by Unanimity
5 Criteria = ["Paper not in the scope", "Papers not
      in English"]
```

Listing 4 Excerpt from the *Mobile app development* protocol written in ReLiS

was an SLR study on mobile app development (Brunschwig et al. 2022). We show an excerpt of its protocol model in Listing 4. The SLR project was generated with two exclusion criteria, shown in the listing (*"Paper not in the scope"*, *"Papers not in English"*). However, during the SLR process, while the SLR app was running, the project manager modified the set of exclusion criteria to adapt the evolving needs of the study. They added three more criteria: *"Paper not the right length: 3 to 20 pages only"*, *"Paper doesn't focus on energy efficiency practices"*, and *"Paper not related to mobile applications directly"*.

Overall, the studied projects collectively encompassed 16 829 papers and 58 exclusion criteria. Using our approach, we manually searched for inconsistencies in these projects and identified inconsistencies in the exclusion criteria of 5 projects when compared to the initial specification.

6.4. Results

We applied the workflow presented in Section 5 to all 10 projects. We constructed the rule mapping model for the projects in a manner where each project is associated with two rules. One of these rules is derived from the exclusion criteria defined in the ReLiS configuration file, while the other is extracted from the metadata within the ReLiS database. Notably, the latter rule could have been modified via the ReLiS portal by project administrators, distinct from changes made in the configuration file. These two rules are interrelated for each project. Our prototype successfully identified the five inconsistent projects and the five consistent ones with respect to exclusion criteria.

As an example, we show an excerpt from the generated Alloy code for the *Mobile app development* project in Listing 5. Our analysis identified this project as inconsistent and identifies an inconsistency regarding exclusion criteria between the protocol model and the database of the runtime data. Specifically, the

```
1 open Consistency
2 one sig mobileApps extends Artifact {}
3 one sig mobileApps_RecordLimit_2 extends Rule {}
4 one sig ref_exclusioncrieria_TypeLimit_2 extends
      Rule {}
5 one sig RelisDm extends Scenario {
6 } {
7  artifacts = ... + mobileApps_RecordLimit_2 + ...
8  givenRM = ... + (mobileApps_RecordLimit_2 ->
      ref_exclusioncrieria_TypeLimit_2) + (
      ref_exclusioncrieria_TypeLimit_2 ->
      mobileApps_RecordLimit_2) + ...
9  givenRE = ... + (mobileApps_RecordLimit_2 ->
      ref_exclusioncrieria_TypeLimit_2) + (
      ref_exclusioncrieria_TypeLimit_2 ->
      mobileApps_RecordLimit_2) + ...
10  givenE =  ...
11 }

13 assert
      mobileapps_RecordLimit_2__exclusioncrieria_
      TypeLimit_2 {
14  Consistency_r[RelisDm, mobileApps, mobileApps,
      mobileApps_RecordLimit_2,
      ref_exclusioncrieria_TypeLimit_2]
15 }
16 check mobileapps_RecordLimit_2__exclusioncrieria_
       TypeLimit_2 for 1
```

Listing 5 Excerpt from the generated Alloy code for the *Mobile app development* project

assertion defined in Lines 13–15 of Listing 5 was shown by Alloy to be invalid. This is because the rule in Line 3 and the updated rule in Line 4 are evaluated differently for the initial specification of the project and runtime data.

This gives us evidence to answer **RQ1** by affirming that our approach can correctly detect inconsistencies in real low-code applications.

6.5. Threats to validity

The validity of our case study is threatened by the choice of the low-code platform and the choice of projects. This is acceptable because we are not claiming general applicability of our approach. Instead, we aim to make the argument that it is *possible* to use our approach in real applications to correctly produce useful results.

Another threat to validity is the relatively small size of the

low code applications (SLR projects) that we studied. To mitigate this, we present a second evaluation in the next section, where we experimentally study the scalability, sensitivity and efficiency of our approach for increasing sizes of the various aspects of the consistency detection scenarios.

7. Performance Evaluation

The ReLiS case study involved a small set of artifacts and simple rules. We now report on experiments we conducted on a larger dataset using synthetic data to evaluate the performance of our inconsistency detection approach.

7.1. Objectives

For performance evaluation, we focus on the generation and execution of the Alloy code, because the performance of our approach highly depends on this component. However, the overall performance of the system depends on the performance of artifact-level rule evaluators. Therefore, we evaluate the performance of the Alloy component relative to the performance of the evaluators, to assess whether it impacts the overall system performance. Our proposed consistency checking approach is meant to complement existing artifact-level rule checking already present in LCDPs. So it should not impose an undue reasoning burden, relative to any existing rule verification load in a given LCDP.

We focus on the following research questions:

RQ2: Sensitivity – What is the impact of rule mapping complexity on our approach? We hypothesize a correlation between the execution time of the Alloy code and the complexity of rule mapping.

RQ3: Scalability – How does the size of input data affect our approach? We hypothesize that the input size, quantified by the number of records, exerts minimal influence on our approach.

RQ4: Efficiency – What is the impact of incorporating the inconsistency detection mechanism in a low-code generated application in operation with respect to time? We hypothesize that the incorporation of our approach as a new layer for inconsistency detection places a negligible to acceptable burden on the performance of the system.

7.2. Experimental subjects

In the absence of publicly available LCDP datasets, we opt to use synthetic data to evaluate our approach.

Specifically, to validate our hypotheses, we experiment with a synthesized suite of input data obtained from Microsoft AdventureWorks, a set of datasets published by Microsoft to test and demonstrate the capabilities of Microsoft SQL Server[1]. We use the `Person`, `Phone`, and `PersonPhone` tables from Microsoft AdventureWorks 2019. These tables represent realistic data that one would typically find in a phonebook application generated by a LCDP, and are thus suitable source of synthetic data for our evaluation. Additionally, we formulate simple, yet practical, business rules for such an application. For instance, we established rules like "each person must have N phone numbers" (R1) and "each person must have X home numbers and Y cell numbers" (R2).

To simulate the rule evaluators that would be implemented in a real low-code application, we developed custom evaluators in Java. For example, the evaluator for R1 performs a straightforward count of entries in the `PersonPhone` table associated with a given `Person` entry, ensuring it does not exceed the specified limit. The rule evaluators employ a simple counting procedure to yield a yes/no output. These evaluators represent simple calculations compared to the more intricate calculations that might be necessary for evaluating complex rules. For instance, in scenarios with more sophisticated rule evaluators, substantial resources may be required, coupled with intricate data computations to determine the final binary evaluation. An example of a more complex rule is *"tasks assigned to each employee group should relate to a single topic"* and *"tasks assigned to each employee group must align with their skills"*. These rules call for in-depth calculations and data extraction to deliver the final evaluative verdict; thereby demanding a greater allocation of time and other resources. As explained in Section 7.3.2, comparing the impact of our approach relative to *simple* evaluators allows us to better assess how light is the overall impact of incorporating our additional consistency detection layer to an LCDP.

7.3. Metrics

7.3.1. Independent variables

We create varying dataset sizes by sampling from the entire dataset, and simulate different scenarios similar to SimplePM to create rules. The suite includes datasets with up to 500 000 records and 40 rules, with varying degrees of mapping complexity, ranging from easy to medium and hard. We use the following independent variables:

- Number of records
- Number of rules
- Mapping complexity
- Number of inconsistencies

To explain mapping complexity, in our analysis, we treat the rules and their relationships as a graph. The nodes represent the rules and the edges represent the related rules. The frequency of edge occurrences serves as a quantitative indicator of the complexity degree. As we currently perform the rule mapping step manually, as depicted in Figure 4, we create rule sets with 10, 20, and 40 rules to simulate increasing difficulty in the mapping phase.

We construct rule graphs to represent a spectrum of easy to hard mapping complexity levels. For every mapping, we generate graphs with the following number of edges calculated based on n, where n represents the number of rules.

$$\text{Number of edges} = \begin{cases} ((n-2)/2)+1 & \text{if } \textit{easy} \\ \lfloor((n-2)/2)^{1.75}\rfloor+1 & \text{if } \textit{medium} \\ ((n-2)/2)^2+1 & \text{if } \textit{hard} \end{cases}$$

Essentially, we parameterize the graph's edge count to generate mappings of diverse complexities. Graphs with a greater number of edges signify highly interconnected rule mappings, while

[1] See: https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/adventure-works

Table 2 Excerpt from the experiment results with synthetic data - Times are in seconds.

	# Records	# Rules	Mapping Complexity	# Inconsistencies	Time in Evaluators	Time in Alloy	ACOR
1	5 000	10	Easy	None	0.034	0.026	43%
2	5 000	10	Medium	One	0.026	0.052	67%
3	5 000	20	Medium	All	0.044	0.422	90%
4	5 000	40	Hard	Some	0.078	37.092	100%
5	50 000	10	Easy	None	0.226	0.023	9%
6	50 000	10	Medium	One	0.218	0.048	18%
7	50 000	20	Medium	All	0.400	0.399	50%
8	50 000	40	Hard	Some	0.726	36.740	98%
9	500 000	10	Easy	None	2.032	0.029	1%
10	500 000	10	Medium	One	2.060	0.050	2%
11	500 000	20	Medium	All	5.103	0.406	7%
12	500 000	40	Hard	Some	7.827	36.830	82%

those with fewer edges indicate lower interdependence between rules. We use a power of 1.75 for medium mappings to ensure a distinctive growth rate compared to the other two mapping complexities.

In our experimental setup, we implemented straightforward Java-based external rule evaluators based on the modular definition of consistency in Section 4 that allows outsourcing the evaluation of artifact-level rules to artifact-specific reasoners. This approach facilitates fast and efficient evaluation, simulating the optimized checks typically found in real LCDPs. Additionally, it enables us to manually set the evaluation results to simulate various levels of inconsistency, ranging from no inconsistencies to all rules being inconsistent.

The inclusion of $(n-2)/2$ in our calculations of mapping complexity is due to our use of bipartite graphs. We use bipartite graphs to facilitate the simulation of differing levels of inconsistency and mapping complexities. Within our approach, inconsistency is deduced through the concurrence of evaluation outcomes in two interconnected rules. Consequently, instances where two YES or two NO results emerge are deemed as consistent scenarios.

In our testing process, we replicate varying degrees of inconsistency, encompassing none, one, some, and all. Figure 8 shows a case of hard mapping complexity involving eight rules. We make it so that the first two rules are always related in our graphs, allowing us to easily recreate the scenario of *one* inconsistency, by altering the evaluation outcome of one rule to NO and the other to YES. To recreate the scenario of *all* rules being inconsistent, we make the odd-numbered nodes be YES and the even-numbered nodes NO. To recreate the scenario of *some* inconsistencies, we designate all nodes as YES, except for half of the odd-numbered nodes.

The above strategies allow us to create experimental subjects of various complexity gradations of the independent variables.

7.3.2. Dependent variables To answer the research questions RQ2-RQ4, we measure the time usage of the Alloy component and the rule evaluators of our approach for varying complexities of input data, rule mappings, and inconsistencies. Furthermore, we aim to quantify the impact of the Alloy component on the operation of the entire system.

Artifact-level rule evaluators are a resource-intensive aspect

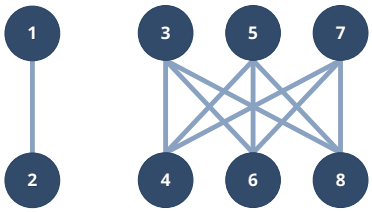

Figure 8 An example of a bipartite graph of eight rules with the "hard" mapping complexity. This graph is unrelated to Figure 3. It corresponds to the auto-generated rule mapping graph for the app-level rules.

of the system. As presented in Section 5, our approach depends on the result of artifact-level rule evaluations within the rule mapping model prior to initiating the Alloy code generation phase (See Figure 4). To capture the efficiency of our approach, we define the *Alloy Component Overhead Ratio* (ACOR) as:

$$ACOR = \frac{\text{Time spent in Alloy}}{\text{Time spent in Alloy} + \text{Time spent in evaluators}} \quad (5)$$

Higher ACOR values indicate that the system's overall performance is degraded by the added overhead of our Alloy-based inconsistency component. Smaller ACOR values mean that it exerts a lesser influence on the overall system performance relative to the performance of the LCDP's own rule evaluators. To underscore the impact of our component in our evaluation, we specifically chose simple implementations of external rule evaluators, to simulate the fast, optimized rule evaluation checks that real LCDPs would be expected to deploy. The similarly fast calculations in our evaluators tend to minimize the value "time spent in evaluators" in the denominator of the ACOR metric. Thus, using simple evaluators allows us to assess the overhead of our approach in the least favourable scenario – a setup with more complex rule evaluators would be more forgiving.

Based on the formalization in Section 4, the value of the "time spent in Alloy" in ACOR is associated with the implementation of $consistency(a_x, b_y)$ in Equation 3 within our prototype. Similarly, the time usage in evaluators is related to the

implementation of $E(a_x, r_n) \Leftrightarrow E(b_y, r_k)$ in Equation 2 within our prototype.

7.4. Experiment setup

The experiments were conducted on a machine running Windows 11, equipped with an AMD Ryzen 7 5700U processor clocked at 1.8GHz, 16GB of RAM, and utilizing Alloy 5 and Eclipse Epsilon with Java 17.0.1, with a heap size set to 8GB.

7.5. Results

We show the results of the experiments in Table 2. To address the research Questions RQ2, RQ3, and RQ4, we summarize the correlations we identified using the Pearson correlation coefficient in Table 3 and Table 4.

7.5.1. Sensitivity (RQ2). The sensitivity analysis of our approach aimed to explore the effect of mapping complexity on our Alloy component. To this end, we conducted several experiments on synthetic data, where we varied the mapping complexity while keeping all other independent variables constant. Our findings indicate that the influence of rule mapping complexity on our approach is substantial, demonstrated by the strong positive correlation with the performance of the Alloy component ($r = 0.95, p = 0.000$), shown in Table 3. This means that more complex rule mappings tend to slow down the performance of the Alloy component.

We can also see the impact of the number of rules on our approach, as indicated by the positive correlation between the number of rules with Alloy component performance ($r = 0.53, p = 0.000$) shown in Table 3.

7.5.2. Scalability (RQ3). The scalability of our approach was evaluated by investigating the effect of the size of the input on our Alloy component. Our experiments showed that the approach is generally scalable, with respect to the number of records.

The impact of input size on our approach is negligible, as indicated by the very weak correlation with Alloy component performance ($r = 0.00, p = 0.999$) shown in Table 3. Our approach displays insensitivity to variations in input size, particularly with respect to the number of records, and number of inconsistencies. This suggests that variations in input size and number of inconsistencies do not significantly affect the performance of the Alloy component, thus the approach is scalable in terms of the input size.

However, the time spent in evaluators tended to increase more rapidly with larger input sizes, indicating that the evaluation process becomes more computationally expensive as the input size grows.

7.5.3. Efficiency (RQ4). We compared the time spent in rule evaluation with the time spent in Alloy to provide insights into the efficiency of the overall approach. Using the Alloy Component Overhead Ratio (ACOR) metric, we studied the added overhead of the inconsistency detection layer. The results are shown in Table 4.

For applications with small and medium number of rules (10 and 20, respectively), we observe a strong negative correlation between ACOR and the number of records ($r = -0.93, p = 0.000$). This indicates that as the input size, measured by the number of records, increases, the ACOR decreases significantly. In other words, as the application handles more data, the Alloy component's relative performance becomes notably better (i.e., less overhead to the whole system), suggesting that incorporating the inconsistency detection mechanism is an efficient choice in these scenarios.

Moreover, for applications with the maximum number of rules (40), we note a substantial positive correlation between ACOR and mapping complexity ($r = 0.62, p = 0.000$). This implies that as the rule mapping complexity intensifies, ACOR also increases significantly. Therefore, when dealing with complex rule mappings, the incorporation of the inconsistency detection mechanism could potentially lead to a noticeable overhead.

7.6. Discussion

Based on the correlations presented in Table 3 and Table 4, along with the results presented in Section 7.5, we can draw insights into the performance of our approach under various conditions.

Our approach exhibits good performance when dealing with less intricate rule relationships and big datasets. The degree of rule complexity and interconnection varies among software applications, and it is essential to consider these characteristics when contemplating the adoption of our approach. In scenarios characterized by a significant volume of data, as indicated by the number of records, and a relatively low degree of rule interconnection, our approach proves to be a valuable choice for detecting inconsistencies.

In summary, our approach's suitability varies based on factors like rule complexity, data size, and the degree of rule interconnection within a given software application. These considerations should guide the decision to incorporate our approach into a specific context.

Currently, we have developed a proof of concept prototype. We envision that a full implementation of our approach would be incorporated into an LCDP either as a plugin or through API interfaces. As previously mentioned, our approach performs well with data-intensive low-code applications characterized by fewer interconnected rules. To achieve integration with an LCDP, we envision providing an assistant tool to aid administrators and app developers in rule mappings. This tool will abstract the complexities associated with instantiating the trace DSL. Additionally, while structured outputs of our consistency checks are available, they require better interpretation and presentation for users. With the inclusion of a mapping assistant tool and the provision of more user-friendly reports in the future, integration of our approach with an LCDP will become feasible.

7.7. Threats to validity

One threat to validity concerns to the construct validity of the Alloy Component Overhead Ratio (ACOR) as a metric to capture the Alloy component's impact on the entire system. It is possible that certain rule types, which are not implemented, remain unaccounted for in ACOR. Moreover, our synthetic data may not entirely emulate the complexity of real-world low-code ap-

Table 3 Pearson correlation test between Alloy Time/Evaluation Time and independent variables. Each cell shows (r-value, p-value).

	Input size (Number of records)	Number of rules	Mapping complexity	Number of inconsistencies
Alloy component	(0.000, 0.999)	**(0.528, 0.000)**	**(0.954, 0.000)**	(0.310, 0.001)
Rule evaluators	**(0.767, 0.000)**	(0.329, 0.001)	(0.202, 0.036)	(0.169, 0.081)

Table 4 Pearson correlation test between ACOR and Numer of records/Mapping complexity for fixed number of rules. Each cell shows (r-value, p-value).

	Input size (Number of records)	Mapping complexity
Minimum number of rules (10)	**(-0.932, 0.000)**	(0.191, 0.264)
Medium number of rules (20)	**(-0.928, 0.000)**	(0.312, 0.064)
Maximum number of rules (40)	**(-0.657, 0.000)**	**(0.624, 0.000)**

plications. Although we have endeavored to generate synthetic data that simulates real-world scenarios, the limited variability within the data could impact the generalizability of our results.

Another potential threat arises from the choice of the Alloy solver and its specific configuration. During our testing, we encountered challenges when dealing with a large number of rules, resulting in excessively large Alloy files that its file parser struggled to handle. A possible solution to this issue could involve preprocessing the text within the file and then feeding the generated code to the Alloy API engine as multiple files, making it more manageable. Additionally, the selection of evaluators for comparison could influence the performance of our approach. To mitigate this threat, we opted for evaluators that are intentionally simple and efficient. It is worth noting that, in real low-code platforms, artifact-level rules can be much more complicated, potentially being as complex as OCL (Object Management Group 2014) expressions.

One threat to the validity of our study is that our synthetic data may not fully capture the complexity of real-world low-code applications. While we have made an effort to generate synthetic data that mimics real-world scenarios, the lack of variability in the data may affect the generalizability of our findings.

Our study focuses on a particular category of low-code platforms, as explained in Section 3.1. Our perspective and understanding of low-code platforms may not necessarily apply to all general-purpose low-code development platforms. Furthermore, Alloy is a formal tool appropriate for prototyping, and our findings are based on this specific implementation. It is important to recognize that our results, which rely on this version of our implementation using Alloy, may not be readily transferable to other formal tools employed for implementing our formalization and detecting inconsistencies. Finally, the performance of our approach may be affected by the choice of the Alloy solver and the specific configuration of the solver.

8. Related Work

In this section, we present a comprehensive overview of the state-of-the-art research. The focus of this review is on four key areas that are related to the research objectives of this paper. We reviewed relevant research on consistency management in software engineering, which has been extensively studied in multiple sub-disciplines. The research was divided into four main focus areas: Round-trip Engineering, Model-Metamodel Co-evolution, Database Schema-Data Co-evolution, and Model Inconsistency Management.

8.1. Round-trip Engineering

In the field of round-trip engineering, low-level artifacts (such as code) are derived from high-level models (such as UML diagrams), and changes made to the generated artifacts are reflected back in the high-level models (Sendall & Küster 2004).

Angyal *et al.* offer a synchronization method that supports bidirectional change propagation between code and models by elevating the concrete syntax of code to the model level, thus allowing indirect manipulation of code through model transformations. Additionally, by incorporating abstract syntax trees into the code generation process, the problem is reduced to a model-to-model synchronization task (Angyal et al. 2008). Paesschen *et al.* identify a set of round-trip engineering scenarios that account for various bidirectional changes made to elements and relationships in both code and models. They address these scenarios using a tool-supported approach, where the entities of the data modeling view and the corresponding implementation objects are the same, synchronizing different views through a shared repository of entities (Paesschen et al. 2005). Yu *et al.* address the inconsistency between generated code and user code bidirectionally by utilizing an invariant traceability framework. They maintain traceability links as artifacts evolve and ensure that changes made to generated methods in code, whether from the code template or the model, are synchronized (Yu et al. 2012). Riedl-Ehrenleitner *et al.* present an approach for identifying inconsistencies between UML models and code. Their method requires user-specified consistency rules and relies on the existence of a single, coherent metamodel for both models and code. They use the user-provided rules and the shared metamodel to assess the consistency of UML models and source code (Riedl-Ehrenleitner et al. 2014). Pham *et al.* propose a bidirectional approach between UML models and code to maintain consistency between the structure of generated code and architectural models. Their method reduces

the abstraction gap between models and code by raising the abstraction level of a programming language, though it is only implemented for one language (Pham et al. 2017).

Despite these efforts, none of these techniques account for consistency-breaking changes that may occur while the system is in operation and impact artifacts at any level of abstraction. Regenerating low-level artifacts may not always be a straightforward task and can lead to data loss or disruption to the system state. As such, these round-trip engineering methods do not adequately address the issues outlined in the previous section.

8.2. Model Metamodel Co-evolution

In MDE, metamodels serve as the formal definition of models and are subject to evolution throughout their lifecycle. However, the evolution of metamodels can pose challenges for existing models that conform to previous versions of the metamodel. One approach to address this issue is to migrate models to align with the changes in the metamodel (Cicchetti et al. 2008). Nevertheless, this process can result in data loss if not properly handled (Narayanan et al. 2009).

Cicchetti *et al.* proposed a method based on metamodel differencing and higher-order transformations to automatically generate co-evolving actions, allowing the designer to refine the changes (Cicchetti et al. 2008). Another approach proposed by Cicchetti *et al.* focuses on the evolution of web applications and uses system model differencing to detect and represent structural changes, generating semi-automated migration support to assist the designer (Cicchetti et al. 2011). Kessentini *et al.* proposed a multi-objective approach using genetic algorithms to create a new model that conforms to the evolved metamodel (Kessentini et al. 2019). Khelladi *et al.* addressed the co-evolution of metamodels and code, proposing a method to detect metamodel changes and analyze their impact on code, involving developers in the resolution process (Khelladi et al. 2020).

The inconsistencies that arise in this research domain are only resolved during the development cycle, not during the operation of the system. It is critical to maintain the consistency between the model and metamodel, especially given their well-defined conformance relationship, throughout the co-evolution process. Additionally, the co-evolution must also consider the horizontal consistency between instances, taking into account semantic and property-based constraints that go beyond the simple syntactic relationship between model and metamodel. Despite the numerous approaches proposed in the literature, the resolution of these inconsistencies remains a challenge, as these methods fail to address several important issues.

8.3. Database Schema Data Co-evolution

In this paper, we delve into the realm of data-intensive applications generated through LCDPs, where the complexities of database schema-data co-evolution come into play. This issue is synonymous with the classic problems of schema evolution (Lin & Neamtiu 2009), data cleaning (Fan et al. 2008), and schema repair (Wijsen 2005) in database research. Lin *et al.* highlight the challenges of application/database co-evolution, particularly in terms of mismatched assumptions between the database and the application that can lead to data loss or compromise data integrity. Through a comprehensive study of two open-source applications, the authors extract the differences between two schema versions and outline the difficulties posed by consistency issues (Lin & Neamtiu 2009). Fan *et al.* tackle the problem of data consistency with the Conditional Functional Dependencies method, an extension of Functional Dependencies that enforces semantically-related constants. The authors demonstrate the efficacy of their approach in detecting and repairing inconsistencies in the data (Fan et al. 2008). Wijsen *et al.* take a novel approach to preserve data consistency, proposing to update the schema rather than repair the data through deletion or insertion. The authors advocate for the creation of a single database that consistently answers queries without the need for rewriting (Wijsen 2005).

In conclusion, a holistic approach to consistency management must consider the lessons learned from classic database schema-data problems. However, it is crucial to remember that in databases, the relationship between the data and the schema is well-defined, with clearly separated meta-levels.

8.4. Model Inconsistency Management

In the realm of model-based systems, multiple languages can be used to model the target system from varying angles, as each collaborator brings their unique point of view (David et al. 2016). This notion is embraced by multi-paradigm modeling (MPM)(Mosterman & Vangheluwe 2004) which advocates for modeling every aspect of the system using the most suitable formalisms and abstraction levels. Such scenarios may result in inconsistencies between the models and views of different users.

Egyed *et al.* introduce an automatic method for identifying inconsistencies in software engineering design models and delivering instant feedback on such inconsistencies when the models change. This method tracks the behavior of consistency rules to assess the impact of model modifications (Egyed 2011). Almeida *et al.* suggest an approach that employs Prolog to repair inconsistencies in UML design models. The objective of this approach is to prepare a plan of action to resolve inconsistencies while minimizing the introduction of new inconsistencies. The size of the space explored to resolve the inconsistencies is adjustable (da Silva et al. 2010). Kolovos *et al.* provide a categorization of the different types of inconsistencies that can occur between models in a model-driven development process. They demonstrate the use of the Epsilon Validation Language for detecting and repairing these inconsistencies (Kolovos et al. 2008).

Although separate techniques can be used to tackle specific problems with models at the same level of abstraction, we advocate for a holistic approach that enables coordination between these techniques. However, incorporating parts of these methods into our setting would require significant modifications to the assumptions and data structures they use. Additionally, these methods are functional in the design phase of the product lifecycle, unlike in generated applications LCDPs. The relationships between artifacts in LCDPs are not well-defined, and existing model inconsistency management methods do not account for the aforementioned problems.

8.5. Novel approach distinctions

The above-mentioned approaches often focus on detecting and resolving inconsistencies at design time, which is not applicable to LCDPs where inconsistencies can arise at runtime. Also, these approaches often assume a high level of technical proficiency from users, which can be problematic as application developers and administrators may not necessarily possess software engineering expertise. Moreover, these techniques do not take into consideration the specific constraints and requirements of LCDP, where inconsistencies may arise from the co-evolution of the model and data during operation. Additionally, the approaches assume there are well-defined and well-structured meta-relationships between artifacts, which is not the case in LCDPs where the metalevels are not clearly separated. Therefore, the existing methods fall short in adequately addressing the consistency challenges (C1–C4) identified in Section 1. Thus, there is a need for novel approaches that are specifically tailored to the unique characteristics of LCDPs and their generated applications, with the aim of effectively managing consistency.

On the other hand, we aim to address the inconsistencies that arise in low-code applications during runtime, considering multiple artifacts at different levels of abstraction, and taking into account the limited technical expertise of the citizen developers who use these platforms. In a low-code setup, it is crucial to address the challenge of changing rules by different stakeholders during the operation of the generated application. This dynamic nature can introduce contradictions between rules. Our approach tackles this issue by defining and checking consistency as relationships between artifacts and their associated rules. This novel solution enables us to identify contradictions by detecting disagreements in artifact-level evaluations. By delegating the responsibility of defining and evaluating artifact-level rules to artifact-specific reasoners, our approach achieves a high level of generality and wide applicability. This is because our framework relies exclusively on the results generated by the reasoners or rule evaluators, which perform straightforward yes/no checks. The implementation of each reasoner or evaluator can be customized and integrated into our framework, offering flexibility and adaptability to different scenarios.

Building on OutSystems and adopting some Model-Driven Engineering (MDE) techniques, we developed and generated an application to test our approach to address the consistency challenge in LCDPs.

9. Conclusion

Inconsistency arising from the co-evolution of models and data can pose challenges in LCDPs. Existing techniques for managing consistency are often insufficient due to the specific characteristics of low-code development. In this paper, we propose an approach for detecting inconsistencies between multiple artifacts at different levels of abstraction in LCDPs and their generated applications.

Our approach leverages the MDE paradigm to represent various elements of LCDPs and their generated applications. We use lightweight formal analysis, based on Alloy to reason about the consistency of the generated application. We provide methodological and tool support[2] for detecting inconsistencies that may arise between multiple artifacts in low-code generated applications at runtime due to model/data co-evolution.

We evaluated the correctness of our approach using a case study of 10 real applications developed on a domain-specific LCDP. We also evaluated the scalability, efficiency, and sensitivity of our approach with experiments on synthetic data. Our evaluation demonstrates that our approach effectively detects inconsistencies while maintaining a desirable level of scalability and efficiency.

In the future, we plan to leverage reverse engineering approaches to enhance the automation of the model extraction process. We aim to address any detected inconsistencies and restore the system to a consistent state. We acknowledge that various uncertainties (Troya et al. 2021) may arise during the restoration process, which require careful consideration. Additionally, we intend to provide facilities for semi-automating the rule mapping step, with the goal of streamlining the overall approach and improving its efficiency.

References

Angyal, L., Lengyel, L., & Charaf, H. (2008). A synchronizing technique for syntactic model-code round-trip engineering. In *International conference and workshop on the engineering of computer based systems*. IEEE. doi: 10.1109/ecbs.2008.33

Balzer, R. (1991). Tolerating inconsistency (software development). In *Proceedings-13th international conference on software engineering* (pp. 158–159).

Bigendako, B., & Syriani, E. (2018). Modeling a tool for conducting systematic reviews iteratively. In *Model-driven engineering and software development* (Vol. 1, pp. 552–559). Scitepress. doi: 10.5220/0006664405520559

Bloomberg, J. (2018). *Low-code/no-code? hpapaas? here's what everybody is missing.* Retrieved from https://www.forbes.com/sites/jasonbloomberg/2018/07/30/low-codeno-code-hpapaas-heres-what-everybody-is-missing/?sh=6c7a22493ac0 (accessed 10-Feb-2024)

Brunschwig, L., Guerra, E., & de Lara, J. (2022). Modelling on mobile devices – a systematic mapping study. *Software & Systems Modeling*, *21*, 179–205. doi: 10.1007/s10270-021-00897-8

Cabot, J. (2023). *Low-code vs model-driven: are they the same?* Retrieved from https://modeling-languages.com/low-code-vs-model-driven (accessed 10-March-2023)

Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *Enterprise distributed object computing conference*. IEEE. doi: 10.1109/edoc.2008.44

Cicchetti, A., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2011, feb). Managing the evolution of data-intensive web applications by model-driven techniques. *Software & Systems Modeling*, *12*(1), 53–83. doi: 10.1007/s10270-011-0193-0

da Silva, M. A. A., Mougenot, A., Blanc, X., & Bendraou, R. (2010). Towards automated inconsistency handling in design models. In *Notes on numerical fluid mechanics and*

[2] https://github.com/aminexplo/Incatch

multidisciplinary design (pp. 348–362). Springer. doi: 10.1007/978-3-642-13094-6_28

Dávid, I. (2019). *A foundation for inconsistency management in model-based systems engineering* (Unpublished doctoral dissertation). Universiteit Antwerpen.

David, I., Syriani, E., Verbrugge, C., Buchs, D., Blouin, D., Cicchetti, A., & Vanherpen, K. (2016). Towards inconsistency tolerance by quantification of semantic inconsistencies. In *Workshop on collaborative modelling in mde* (Vol. 1717, pp. 35–44). CEUR-WS.org.

Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., & Wimmer, M. (2022). Low-code development and model-driven engineering: Two sides of the same coin? *Software & Systems Modeling*, *21*(2), 437–446. doi: 10.1007/s10270-021-00970-2

Egyed, A. (2011). Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, *37*(2), 188–204. doi: 10.1109/tse.2010.38

Fan, W., Geerts, F., Jia, X., & Kementsietsidis, A. (2008, jun). Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems*, *33*(2), 1–48. doi: 10.1145/1366102.1366103

Finkelstein, A. (2000). A foolish consistency: Technical challenges in consistency management. In *Database and expert systems applications: 11th international conference, dexa 2000 london, uk, september 4–8, 2000 proceedings 11* (pp. 1–5).

Gomes, C. P., Kautz, H., Sabharwal, A., & Selman, B. (2008). Chapter 2 satisfiability solvers. In *Handbook of knowledge representation* (pp. 89–134). Elsevier. doi: 10.1016/s1574-6526(07)03002-7

Gurcan, F., & Taentzer, G. (2021). Using microsoft PowerApps, mendix and OutSystems in two development scenarios: An experience report. In *Model driven engineering languages and systems companion.* IEEE. doi: 10.1109/models-c53483.2021.00017

Hebig, R., Khelladi, D. E., & Bendraou, R. (2017). Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, *43*(5), 396–414. doi: 10.1109/tse.2016.2610424

Jackson, D. (2011). *Software abstractions, revised edition logic, language, and analysis*. MIT Press.

Kessentini, W., Sahraoui, H., & Wimmer, M. (2019, feb). Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology*, *106*, 49–67. doi: 10.1016/j.infsof.2018.09.003

Khelladi, D. E., Combemale, B., Acher, M., Barais, O., & Jézéquel, J.-M. (2020). Co-evolving code with evolving metamodels. In *International conference on software engineering.* ACM. doi: 10.1145/3377811.3380324

Khorram, F., Mottu, J.-M., & Sunyé, G. (2020). Challenges & opportunities in low-code testing. In *Proceedings of the 23rd acm/ieee international conference on model driven engineering languages and systems: Companion proceedings* (pp. 1–10).

Kitchenham, B., & Charters, S. (2007, jul). *Guidelines for performing systematic literature reviews in software engineering* (Technical report No. EBSE-2007-01). Keele University and Durham University. doi: https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf

Kolovos, D., Paige, R., & Polack, F. (2008). Detecting and repairing inconsistencies across heterogeneous models. In *Software testing, verification, and validation.* IEEE. doi: 10.1109/icst.2008.23

Lin, D.-Y., & Neamtiu, I. (2009). Collateral evolution of applications and databases. In *ERCIM joint workshops on principles of software evolution (IWPSE) and software evolution (evol).* ACM Press. doi: 10.1145/1595808.1595817

Lucas, F. J., Molina, F., & Toval, A. (2009). A systematic review of uml model consistency management. *Information and Software technology*, *51*(12), 1631–1645.

Maksimov, M., Kokaly, S., & Chechik, M. (2020). A survey of tool-supported assurance case assessment techniques. *ACM Computing Surveys*, *52*(5), 1–34. doi: 10.1145/3342481

Mendix. (2023). *Low-code application development platform.* Retrieved from www.mendix.com (accessed 10-March-2023)

Mens, T., & Van Der Straeten, R. (2007). Incremental resolution of model inconsistencies. In *Lecture notes in computer science* (p. 111–126). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-71998-4_7 doi: 10.1007/978-3-540-71998-4_7

Mosterman, P. J., & Vangheluwe, H. (2004). Computer automated multi-paradigm modeling: An introduction. *SIMULATION*, *80*(9), 433–450. doi: 10.1177/0037549704050532

Narayanan, A., Levendovszky, T., Balasubramanian, D., & Karsai, G. (2009). Automatic domain model migration to manage metamodel evolution. In *Model driven engineering languages and systems* (pp. 706–711). Springer. doi: 10.1007/978-3-642-04425-0_57

Object Management Group. (2014). *The Object Constraint Language v.2.4 Specification.* Retrieved from http://www.omg.org/spec/OCL (accessed 10-March-2023)

OutSystems. (2023). *Low-code high-performance software development.* Retrieved from www.outsystems.com (accessed 10-March-2023)

Paesschen, E. V., Meuter, W. D., & D'Hondt, M. (2005). SelfSync: A dynamic round-trip engineering environment. In *Model driven engineering languages and systems* (pp. 633–647). Springer. doi: 10.1007/11557432_47

Pham, V. C., Radermacher, A., Gerard, S., & Li, S. (2017). Bidirectional mapping between architecture model and code for synchronization. In *International conference on software architecture.* IEEE. doi: 10.1109/icsa.2017.41

Reder, A., & Egyed, A. (2012). Incremental consistency checking for complex design rules and larger model changes. In *Model driven engineering languages and systems: 15th international conference, models 2012, innsbruck, austria, september 30–october 5, 2012. proceedings 15* (pp. 202–218).

Riedl-Ehrenleitner, M., Demuth, A., & Egyed, A. (2014). Towards model-and-code consistency checking. In *Annual computer software and applications conference.* IEEE. doi: 10.1109/compsac.2014.91

Rose, L. M., Paige, R. F., Kolovos, D. S., & Polack, F. A. C. (2008). The epsilon generation language. In *Model driven architecture – foundations and applications* (pp. 1–16). Springer. doi: 10.1007/978-3-540-69100-6_1

Sahay, A., Indamutsa, A., Di Ruscio, D., & Pierantonio, A. (2020, aug). Supporting the understanding and comparison of low-code development platforms. In *Euromicro conference on software engineering and advanced applications.* IEEE. doi: 10.1109/seaa51224.2020.00036

Sendall, S., & Küster, J. (2004). Taming model round-trip engineering. In *Best practices for model-driven software development.*

SQLMaestro. (2023). *Database management and web development tools.* Retrieved from www.sqlmaestro.com (accessed 10-March-2023)

Troya, J., Moreno, N., Bertoa, M. F., & Vallecillo, A. (2021, jan). Uncertainty representation in software models: a survey. *Software & Systems Modeling*, *20*(4), 1183–1213. doi: 10.1007/s10270-020-00842-1

Vanherpen, K., Denil, J., David, I., Meulenaere, P. D., Mosterman, P. J., Torngren, M., ... Vangheluwe, H. (2016). Ontological reasoning for consistency in the design of cyber-physical systems. In *Workshop on cyber-physical production systems.* IEEE. doi: 10.1109/cpps.2016.7483922

Wijsen, J. (2005). Database repairing using updates. *ACM Transactions on Database Systems*, *30*(3), 722–768. doi: 10.1145/1093382.1093385

Wong, J., Iijima, K., Leow, A., Jain, A., & Vincent, P. (2023). *Gartner magic quadrant for enterprise low-code application platforms.* Retrieved from https://www.gartner.com/en/documents/4005939 (accessed 10-March-2023)

Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., & Montrieux, L. (2012). Maintaining invariant traceability through bidirectional transformations. In *International conference on software engineering.* IEEE. doi: 10.1109/icse.2012.6227162

Zaheri, M. (2022, oct). Towards consistency management in low-code platforms. In *Proceedings of the 25th international conference on model driven engineering languages and systems: Companion proceedings.* ACM. doi: 10.1145/3550356.3558510

Zaheri, M., Famelis, M., & Syriani, E. (2021, oct). Towards checking consistency-breaking updates between models and generated artifacts. In *2021 ACM/IEEE international conference on model driven engineering languages and systems companion (MODELS-c).* IEEE. doi: 10.1109/models-c53483.2021.00063

About the authors

MohammadAmin Zaheri is a Ph.D. student at the Department of Computer Science and Operations Research (DIRO) of the Université de Montréal, working in the GEODES Software Engineering Research Group. You can contact the author at zaherimo@iro.umontreal.ca or visit http://zaheri.info.

Michalis Famelis is an associate professor at the Department of Computer Science and Operations Research (DIRO) of the Université de Montréal, working in the GEODES Software Engineering Research Group. He is interested in uncertainty, variability, and decision rationale in early design and analysis. You can contact the author at famelis@iro.umontreal.ca or visit http://www.iro.umontreal.ca/~famelis/.

Eugene Syriani is a full professor at the Department of Computer Science and Operations Research (DIRO) of the Université de Montréal, working in the GEODES Software Engineering Research Group. His main research interests fall in software design based on the model-driven engineering approach, the engineering of domain-specific languages, model transformation and code generation, simulation-based design, collaborative modeling, and user experience. You can contact the author at syriani@iro.umontreal.ca or visit http://www.iro.umontreal.ca/~syriani.