applied sciences

MDPI

Article

Combining Rigorous Requirements Specifications with Low-Code Platforms to Rapid Development Software Business Applications

Pedro Galhardo [1,*] and Alberto Rodrigues da Silva [2,*]

[1] Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisbon, Portugal
[2] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisbon, Portugal
* Correspondence: pedromgalhardo@tecnico.ulisboa.pt (P.G.); alberto.silva@tecnico.ulisboa.pt (A.R.d.S.)

Abstract: Low-code development platforms have gained popularity as an effective solution to address urgent market demands for software applications. These platforms have often overcome challenges faced by traditional software development processes, including requirements engineering processes, as they tend to incorporate the requirements in their prototyping phase. However, low-code platforms have followed different approaches with proprietary languages, which is a problem when customers need to move to other technologies or intend to define the specification of their applications in a readable and platform-independent way. To mitigate these challenges, this article discusses a model-driven approach that semi-automatically produces software business applications by combining rigorous requirement specifications (defined with the ITLingo ASL language) with a concrete low-code platform (Quidgest Genio). First, we analyse the common concepts in both ITLingo ASL and Genio languages. Then, we discuss model-to-model transformations that allow converting ASL specifications into Genio low-code projects. Finally, the code generation capabilities of the Genio low-code platform are employed to generate the source code of the target software applications. To evaluate the consistency of the proposed approach, we use and discuss a simple and representative case study based on a fictitious system, the Invoice Management System (IMS), whose requirements are similar to those found in many business applications.

Keywords: requirements specification; model-driven engineering; low-code platforms; ITLingo ASL; Quidgest Genio

Citation: Galhardo, P.; Silva, A.R.d. Combining Rigorous Requirements Specifications with Low-Code Platforms to Rapid Development Software Business Applications. *Appl. Sci.* **2022**, *12*, 9556. https://doi.org/10.3390/app12199556

Academic Editors: Christos Bouras and Peng-Yeng Yin

Received: 26 June 2022
Accepted: 20 September 2022
Published: 23 September 2022

1. Introduction

The development of modern software applications is a complex process requiring a multidisciplinary team. The diversity of backgrounds among the members of such teams causes communication barriers and difficulties in sharing a common vision of the systems to be developed [1]. Requirements engineering (RE) practices help mitigate these barriers' impacts and increase software development teams' productivity [2]. RE has proven to be a crucial collaboration facilitator [3], as business requirements specifications are defined in natural languages (or controlled natural languages) that are easier to understand than programming languages.

Low-code development platforms (or just "low-code platforms" for brevity) started to grow as the need to create applications that quickly adapt to urgent market demands increased [4–6]. Low-code platforms intend to reduce the development and maintenance effort required to deliver and operate some application classes and enable digital-savvy citizen developers (who lack or have limited programming experience) to contribute directly to the software development process [7]. Considering a spectrum where controlled natural languages and programming languages are opposite ends, low-code platform languages stand somewhere in the middle; most of them allow for specifying business requirements more comprehensively than programming languages do. However, they are

not as abstract as platform-independent specification languages. Low-code platforms such as Quidgest Genio [8], OutSystems [9], or Mendix [10] can be particularly beneficial at the prototyping stages of the development process, as experienced users of these tools can create simple yet fully functional systems in a matter of days or weeks. Currently, most large cloud providers offer low-code platforms within their cloud-based solutions. For instance, Microsoft released its Power Apps framework [11] in November 2016; Google acquired the provider AppSheet [12] and made it its primary low-code solution in January 2020; and Amazon released Honeycode [13], a low-code platform for web and mobile application development, in June 2020.

On the other hand, model-driven engineering (MDE) is an approach that considers models not just as documentation artefacts but also as first-class citizens, where models might be used throughout all engineering disciplines and in any application domain [14]. As MDE has similar goals to low-code development, there is an open debate on how low-code development differs from MDE and to what extent work carried out in the field of MDE can be directly transferable to low-code platforms [15]. For instance, Di Ruscio et al. discuss the commonalities and differences between both approaches, concluding that not all model-driven techniques aim at reducing the amount of code needed to implement software solutions, and not all low-code approaches are model-driven [7].

In the scope of this debate, our research intends to show and discuss the combination of a platform-independent language for the specification of software applications, ITLingo ASL [3], with a concrete low-code software development platform, Quidgest Genio [8]. First, we discuss how ASL compares to the Genio language to evaluate the possibility of automatically developing mechanisms to transform ASL specifications into Genio projects. Then, we take the findings of this analysis and design the proposed transformation, represented by the ASL2LC task in Figure 1. It is out of the scope of this paper to discuss the second transformation illustrated in Figure 1 as the LC2Code (code generation) task. Indeed, this transformation has been extensively implemented and evaluated in hundreds of applications of the Genio framework [16].

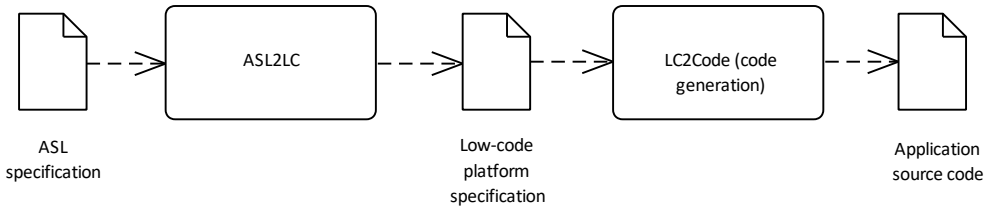

Figure 1. Transforming an ASL specification into a software application via a low-code platform (BPMN notation).

ITLingo is a research initiative that has proposed new languages, tools, and techniques to support users in improving their practices mainly related to project management, requirements engineering, and system design. This initiative has since expanded to include both concluded and ongoing research projects to address challenges like poor productivity and low-quality technical documentation [17].

In this scope, ASL ("Application Specification Language") is research that explores specification languages for the IT domain, mainly targeting the design of software applications [3]. Besides continuously improving as a platform-independent specification language, recent work surrounding ASL has explored model-to-model and model-to-code transformations, respectively, from ASL into ASL specifications and from ASL specifications into Django artefacts [3].

The main contribution of this paper is the proposal and discussion of a functional approach that combines model-driven techniques with low-code platforms. This approach allows developers to quickly write software application specifications in a platform-independent language mainly focused on these applications' data, user interface, and business aspects. ASL specifications can later be automatically converted to concrete low-

code platform specifications to take advantage of the numerous features offered by these platforms. Our work differs from the work developed by Gamito and Silva [3] because, in their approach, they discuss model-to-code transformations, i.e., directly from ASL specifications into Django/Python code. In contrast, our approach is based on the combination of model-to-model and model-to-code transformations, as suggested in Figure 1. Moreover, this paper presents and compares the metamodels underlying both ASL and Genio languages and proposes ASL extensions (developed during this research) to better support the alignment and transformation between ASL and Genio specifications or models.

The rest of this article is organised as follows: Section 2 provides the background on domain-specific languages for the specification of software applications and low-code platforms. Section 3 compares the common concepts of ITLingo ASL and Quidgest Genio languages. Section 4 presents the extensions we propose to ASL based on the previous analysis and comparison findings. Section 5 overviews the ASL2Genio transformation mechanism. Section 6 illustrates how the evaluation and testing of this research were conducted based on a simple yet representative business application. Section 7 identifies and discusses the related work. Finally, Section 8 presents the conclusion, discusses the advantages and limitations of the proposed approach, and identifies future research goals.

2. Background

This section introduces the main concepts and technologies underlying this research. Namely, it introduces ITLingo ASL, a platform-independent language for the specification of software applications, and Quidgest Genio, a low-code platform for developing business software applications.

2.1. Domain-Specific Languages for Business Apps

Domain-specific languages (DSLs) are visual or textual languages specialised in solving problems for specific domains, as opposed to general-purpose languages (GPLs), which provide mechanisms to solve problems in any domain [18]. It is important to note that a higher degree of specialisation means that new DSLs can be created to solve specific problems, and DSLs can solve problems related to their domain faster than GPLs can [19]. Popular examples of DSLs are HTML (Hypertext Markup Language) [20] and CSS (Cascading Style Sheets) [21] (for web pages), or LaTeX (for technical documentation) [22].

In this study, we focus on DSLs for the specification of business applications. A problem faced by development teams in the IT industry is the language barrier between software developers (responsible for building such applications) and business analysts or domain experts (responsible for specifying the business and user requirements). These DSLs can contribute [18,23] to (1) increasing the developer's productivity because they are simpler to understand and write, (2) reducing the language barrier between different stakeholders, and (3) increasing the overall productivity of the development team.

2.2. Low-Code Development Platforms

Low-code development platforms started to grow as the need to create applications that quickly adapt to urgent market demands increased [4–7,24]. These platforms produce applications whose source code is mostly automatically generated rather than being hand-coded as it occurred traditionally. Based on code generation features, this approach aims to reduce delivery times and the size of the teams required to build them. On average, low-code platforms can decrease the time to produce a fully functional system from months to days, or from years to weeks, depending on the complexity of the needs of their customers [25].

The reduced need for manual code allows professionals without an IT background to design and build complex systems using their domain expertise to define application specifications, which may be used to automatically generate fully functional software products. For instance, a study conducted by Gartner in June 2021 predicts that non-IT professionals will build around 80% of technology products and services by 2024 [26]. This

trend is fuelled by the growth of low-code software development platforms and the use of artificial intelligence applied to the automatic generation of code [27].

Moreover, the "low-code" trend is not exclusive to the industry. Some academic research has been conducted towards analysing the evolution of low-code platforms, focusing on their promises and limitations. For example, Frank et al. [5] perform an exploratory study in the market of low-code platforms to determine whether and to what extent the promises and prospects of low-code platforms are appropriate. Their study did not find evidence that low-code platforms go beyond the state of the art in the research. However, the authors acknowledge that the "low-code" trend raises awareness of the importance of researching alternative representations to code. Overeem [6] discusses the evolution of low-code platforms and how they can support the new generation of digital companies. For instance, the author discusses how software applications generated by low-code platforms must include features of modern software systems, such as event sourcing [28], API management [29], and evolution supporting architecture [30], in order to be an effective tool to support the development of complex software systems. Bock et al. [4] provide a balanced account of the current trend of low-code platforms. The authors discuss how low-code platforms compare with the current status of research, and what opportunities for future research arise from the present attention to low-code platforms. Similarly, their analysis did not provide evidence that the individual components of low-code platforms are radical innovations but that the momentum generated by the "low-code" trend gives rise to other significant research opportunities.

2.3. ITLingo ASL

ITLingo ASL (or just ASL for brevity) [3] is a specification language to define software business applications and is part of the ITLingo Initiative [17]. ASL combines some concepts from ITLingo RSL and OMG IFML languages. ITLingo RSL is a requirement and test specification language [31–34]. OMG IFML language is a modelling language to describe the user interface aspects of an application front-end [35].

The fundamental concepts (relevant for this paper) provided by ASL are (1) data entities and enumerations, which represent domain-specific concepts; (2) UI elements, such as UI containers, components, and parts; (3) actors, who represent roles played by users or other systems; and (4) use cases, which represent interactions between the actor(s) and the system under consideration. These concepts are organised in the following key architectural views: data view and use case view (also based on the RSL language) and user interface view (inspired by the IFML language).

To assist the writing of *SL specifications (namely, ASL specifications), some software tools have been offered within the context of the ITLingo Initiative. Currently, authoring ASL specifications is best supported by ITLingo-Studio, a specialised tool for rigorously writing *SL specifications [17]. However, a web version of ITLingo-Studio is being developed, named ITLingo-Cloud. This platform aims to bring multi-organisation and multi-project collaboration features to the ITLingo ecosystem. Figure 2 shows some of the features provided by ITLingo-Studio to aid the specification process, such as syntax highlighting and error checking.

2.4. Quidgest Genio

Quidgest Genio (or just Genio for brevity) is a low-code software development platform developed by Quidgest, a multinational technology company based in Portugal [36]. Genio is defined as an extreme low-code platform (rather than simply a low-code platform) since, according to Quidgest, solutions generated by Genio are composed of, on average, only 2% of manual code [8]. Since most of the source code is automatically generated, the resulting systems are easily maintainable and adaptable to new market demands.

Developing software systems in Genio follows a model-driven engineering approach; most of the effort is put into creating models that can be used by any business process, rather than following a process-driven approach by specifying distinct business processes.

Genio acts as an IDE that supports editing data entities, business rules, forms, and menus of software applications rather than source code. Genio's user interface is shown in Figure 3, including a menu with the main concepts that can be defined.

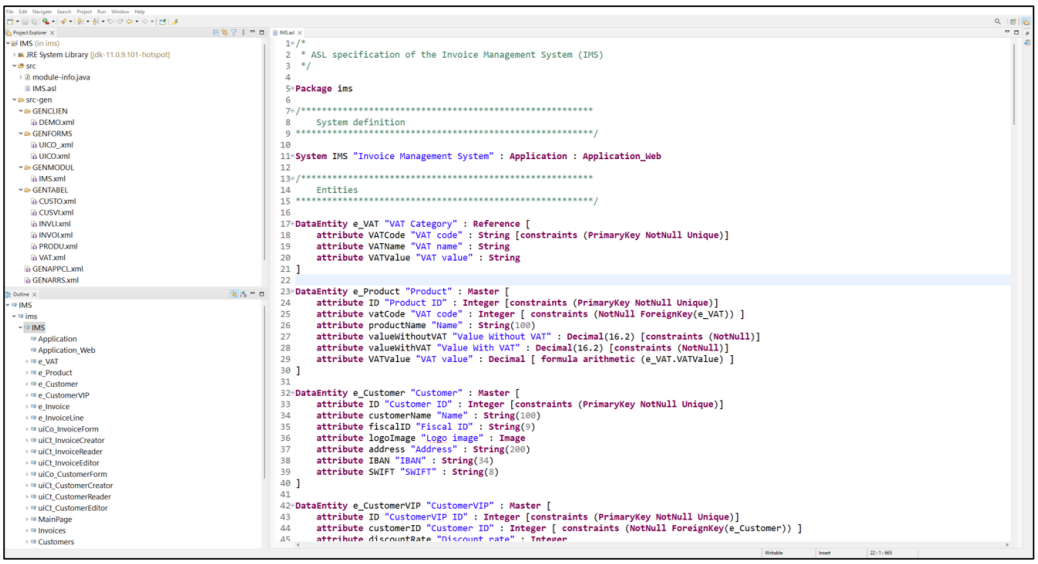

Figure 2. ITLingo-Studio development environment for ASL.

Figure 3. Quidgest Genio's graphical interface.

3. Comparison of ASL and Genio Languages

This section compares and discusses ASL and Genio based on their language metamodels to draw initial conclusions about their closeness and alignments. This comparison aims to identify: (1) concepts that are defined in both ASL and Genio so that the ASL2Genio generator can transform them directly; and (2) concepts that are defined in Genio but not in ASL, so we can design extensions to ASL and later update the generator accordingly.

This analysis focuses on the following groups of concepts usually found in the specification of business applications: data entities, user interface (UI), actors, use cases, and related concepts.

A fictitious "Invoice Management System" (IMS) supports our analysis and discussion. The following text presents a summary description of the IMS informal requirements, which are further detailed in [32,34]:

> *"The Invoice Management System (IMS) is a system that allows users to manage customers, products, and invoices. An IMS user has a user account and is assigned to user roles, such as operator, manager, and customer. For each customer, the system shall maintain the following information: name, fiscal id, logo image, address, bank details, and additional information such as basic personal contact information. Each product must have only one VAT category and maintain the respective current VAT value. Additionally, the system should maximise the productivity of its users by using computed fields, so the users only need to focus on what is strictly necessary. [...]"*

3.1. Data Entities

The elicitation and representation of data entities are usually defined as the first steps of the development process of model-driven business applications (in parallel with use cases or scenarios specification). Data entities denote the key concepts used in domain modelling, data modelling, or data specification [32,37].

Figure 4 shows a fragment of ASL metamodel focused on data specification, which includes DataEntity, DataAttribute, and Constraints concepts. A DataEntity denotes a data element defined by several DataEntityConstraints and DataAttributes. A DataAttribute represents a data value of a particular type (e.g., Integer or String) and is characterised by several DataAttributeConstraints. A DataAttributeConstraint can be used to mark a DataAttribute as a primary key of a DataEntity, to reference other DataAttributes denoting foreign keys, etc.

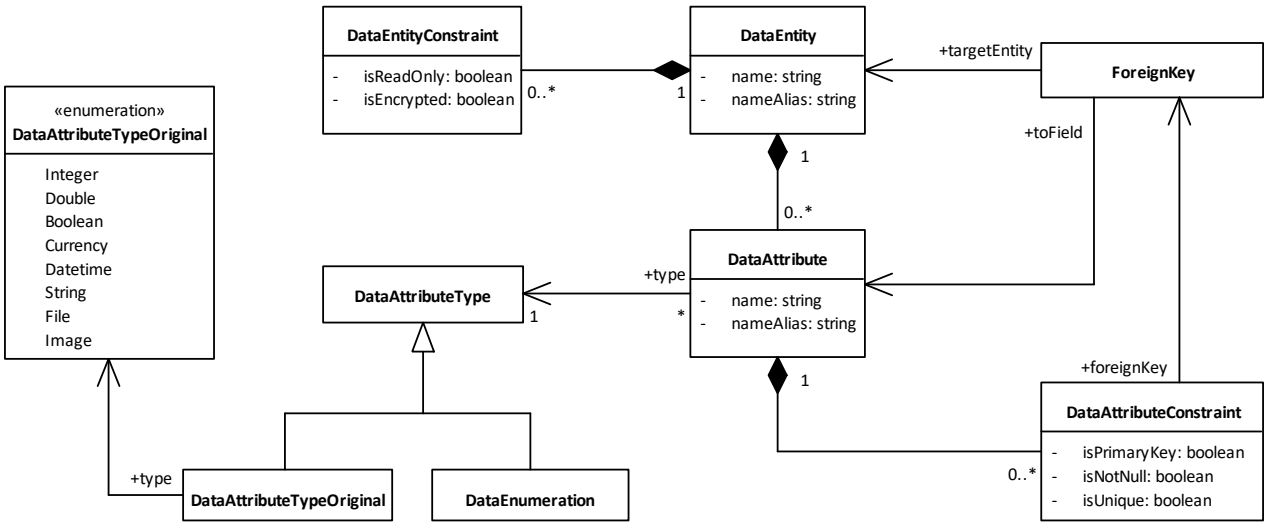

Figure 4. Partial ASL metamodel with data entities-related constructs (UML notation).

Spec. 1 shows a partial ASL specification of data entities that support the IMS system, considering some of the usual properties an invoice (and invoice lines) shall include.

On the other hand, Figure 5 shows a fragment of Genio's metamodel focusing on data entities and related constructs based on Genio's terminology. A Table aggregates a set of TableFields and WriteConditions. TableFields are similar to ASL's DataAttributes, representing a data value with a particular data type. However, Genio supports the specification of different types of formulas, which allow these values to be automatically computed. TableFields can also reference Enumerations, which aggregate a set of Enumeration elements uniquely identified by their id and include a user-friendly description. Finally, WriteConditions are logical expressions that can be defined to enforce business rules at the Table level. For instance, considering the data attribute "Value Without VAT" of the data

entity "Invoice", a WriteCondition can be employed to validate if this value is greater than or equal to zero. If the condition is not met, the record cannot be updated.

```
DataEntity e_Invoice "Invoice" : Document [
attribute ID "Invoice ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute customerID "Customer ID" : Integer [constraints (NotNull ForeignKey(e_Customer))]
attribute dateCreation "Creation Date" : Date [defaultValue "today" constraints (NotNull)]
attribute dateApproval "Approval Date": Date
attribute datePaid "Payment Date" : Date
attribute dateDeleted "Delete Date" : Date
attribute isApproved "Is Approved" : Boolean [defaultValue "False"]
attribute totalValueWithoutVAT "Total Value Without VAT" : Decimal(16.2) [constraints
(NotNull)]
attribute totalValueWithVAT "Total Value With VAT" : Decimal(16.2) [constraints (NotNull)]
attribute totalInvoiceLines "Total invoice lines": Integer
]

DataEntity e_InvoiceLine "InvoiceLine" : Document [
attribute ID "InvoiceLine ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute invoiceID "Invoice ID" : Integer  [constraints (NotNull ForeignKey(e_Invoice))]
attribute productID "Product ID" : Integer [constraints (NotNull ForeignKey(e_Product))]
attribute order "InvoiceLine Order" : Integer [constraints (NotNull)]
attribute valueWithoutVAT "Value Without VAT" : Decimal
attribute valueWithVAT "Value With VAT": Decimal
]
```

Spec. 1. ASL specification of the "Invoice" and "InvoiceLine" data entities (IMS example).

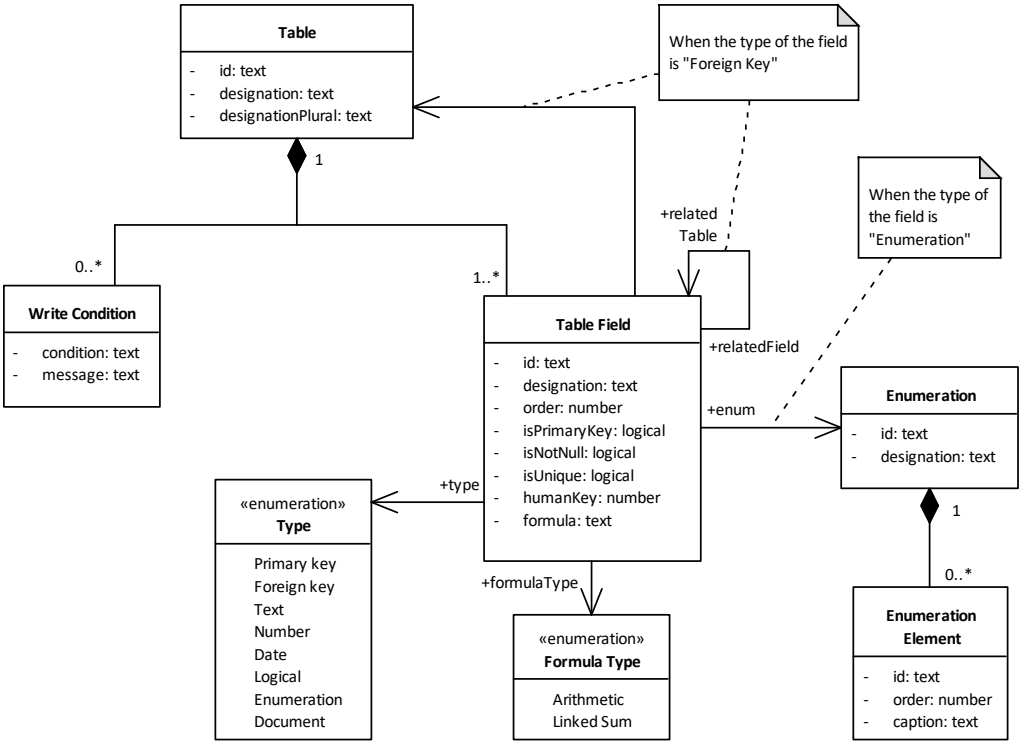

Figure 5. Partial Genio metamodel with data entities-related constructs (UML notation).

By examining how common concepts are expressed in both languages, we can conclude the similarity between these two metamodels. For instance, some aspects, such as identifiers, titles, or constraints, are described similarly in both languages. However, concepts like a field's type are not directly translatable, requiring additional mapping between the representations in both languages. Finally, some minor aspects (such as the plural form of a field's title) can be defined in Genio but not in ASL, and there are some limitations in Genio specifications when compared to ASL, such as the character limit imposed on the definition of data entity and data attribute identifiers.

Still, performing an in-depth analysis focused on data attributes reveals key differences between them that our solution shall take into consideration:

Primary keys. In ASL, the concept of a primary key is defined as a constraint of the data attribute, whereas Genio implements it as a data attribute type. In ASL, it is possible to specify different data entities that use distinct data attribute types for their primary key data attributes, while Genio enforces that the same data attribute type is used for all primary keys. This is a limitation in Genio, as the optimal data attribute type to be used as a data entity's primary key depends on its most common use cases. For instance, using integer primary keys reduces the size of the database, while using GUID primary keys improves the generation of random new keys.

Data enumerations. Regarding the definition of data enumerations, ASL can be improved since it allows only the text values to be specified, while Genio data enumerations have keys associated with the values. For example, the value "Invoice" of data enumeration "Document type" could have an additional key such as "I". These keys could be used for internal use only (never exposed to the application users), such as to reduce the size of the data stored in the database or to uniquely identify the selected value when using multi-language systems.

Formulas. ASL supports the definition of whether a data attribute's value is derived but offers no practical way of specifying how to compute it. In Genio, a field is implicitly derived when its value is computed using a formula. Formulas are expressions (written using Genio's general purpose expression language) evaluated at different moments of the application runtime. Commonly used types of formulas are arithmetic formulas, which can evaluate logical expressions to compute a field value.

Write conditions. As discussed, Genio supports the specification of write conditions, which are business rules written in the form of logical expressions (using Genio's general purpose expression language) that must be enforced when modifying the value of a given record. These expressions are commonly present in Genio data models. ASL, on the other hand, currently does not support the definition of such conditions.

Human-readable keys. Genio uses a property called "human key" to mark a field of a given table as one of the most suitable to represent the record on the user interface (there may be multiple fields marked as human keys). For instance, a primary key is an adequate field to uniquely identify a record during internal operations. Still, it does not provide much information to the application users when displayed on the screen. For that purpose, Genio uses fields that are marked as human keys. For example, the fields "Name" or "Fiscal ID" are potential candidates to be marked as a human key for the table "Person". In ASL, this type of specification is not yet possible.

3.2. UI Elements

Business software applications depend on user input to support data management tasks that build most of the knowledge of these systems. UI elements, such as menus and forms, are commonly defined to support CRUD (i.e., short for "create, read, update, and delete") and other related operations [38,39].

For the specification of UI elements, ASL follows the IFML terminology, in which the UI structure is defined with UI containers, UI components, and UI parts. The rules for expressing such elements are based on the IFML (Interaction Flow Modeling Language) [34]. UI components supported by ASL include lists, details, forms, dialogues, and menus [3,35]. Figure 6 shows a fragment of ASL's metamodel with a focus on UI components and other constructs directly related to them. A UIContainer aggregates a set of UIComponents, and a UIComponent is bound to a given DataEntity and aggregates UIComponentParts. A UIComponentPart is bound to a given DataAttribute and can be defined as a field (i.e., a part that is visible to the user, may trigger events, and may receive values through parameter passing) or a slot (i.e., a value placeholder that is not visible to the user).

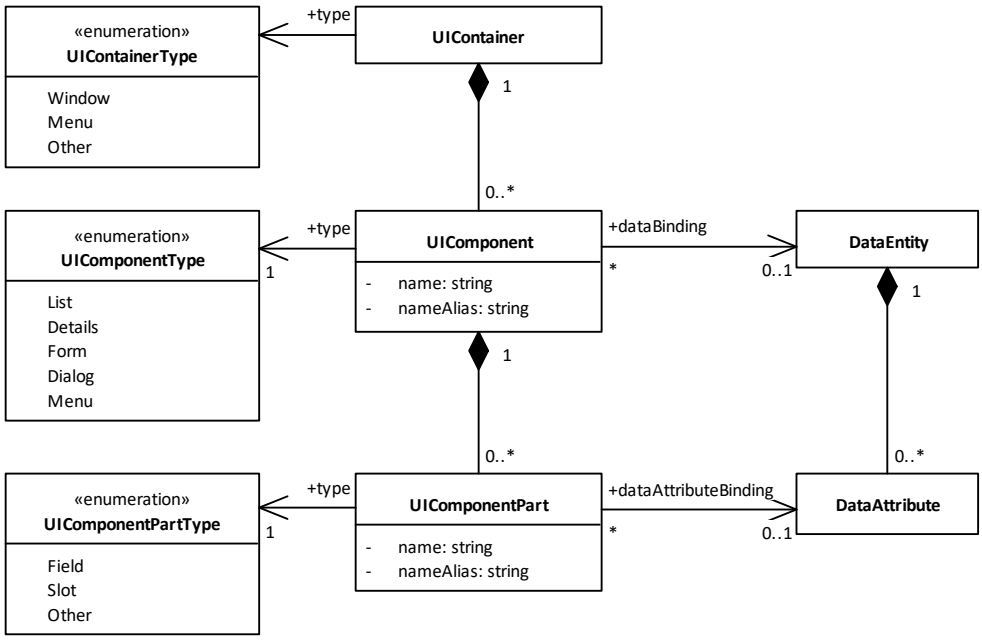

Figure 6. Partial ASL metamodel with UI elements (UML notation).

As suggested by the UIComponentType enumeration, a UIComponent can be used to specify diverse types of UI elements, such as Forms or Lists, which are both directly offered by Genio. Spec. 2 shows an example specification of a Form and a List bound to the "Invoice" DataEntity.

3.2.1. UI Forms

Information systems often use forms to support CRUD operations on data entities. Figure 7 shows a fragment of Genio's metamodel focused on forms and other related constructs. For instance, it shows that Forms shall be bound to a given Table and aggregate a set of Form fields, each bound to a given Table field. In Genio, as suggested in Figure 7, a Form supports the definition of the minimum access level (MAL) required to read and update the record. This means that when a user attempts to open a form to perform a certain action (for example, to update a record), the system verifies if the user's access level is greater or equal to the minimum access level allowed for the corresponding task (in this case, to update the record).

Regarding Genio Forms, we can create a direct mapping to ASL. In particular, ASL's UIComponent may be directly used to represent a Genio form's key concepts without requiring any expansion. For example, the form itself may be defined as a UIComponent, and its fields each as a UIComponentPart.

3.2.2. UI Lists

Like Forms, Lists are UI components commonly found in information systems optimised for data reading, searching, and filtering purposes. Lists usually show data in tabular views and help users deal with large amounts of data.

In Genio, lists are implemented as a type of menu entry. Additionally, Genio supports the definition of lists as form fields, allowing them to be rendered inside forms, which helps list-related records. Figure 8 shows a fragment of Genio's metamodel focusing on menu pages and other related constructs.

```
UIContainer Invoices “Invoices”: Window [
component InvoiceList : List : List_Table [
dataBinding e_Invoice [ orderBy e_Invoice.dateCreation DESC ]

part uip_Customer “Customer” : Field : Field_Output
[ dataAttributeBinding e_Customer.customerName ]
part uip_dateCreation “Creation Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.dateCreation ]
part uip_dateApproval “Approval Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.dateApproval ]
part uip_datePaid “Payment Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.datePaid ]
part uip_dateDeleted “Delete Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.dateDeleted ]
part uip_totalValueWithoutVAT “Total Value Without VAT” : Field : Field_Output
[ dataAttributeBinding e_Invoice.totalValueWithoutVAT ]
part uip_totalValueWithVAT “Total Value With VAT” : Field : Field_Output
[ dataAttributeBinding e_Invoice.totalValueWithVAT ]
]

component uiCo_Filter_Invoice : Details [ dataBinding e_Invoice ]
component uiCo_Search_Invoice : Details [ dataBinding e_Invoice ]

component uiCo_Actions : Menu [
event ev_read “View Invoice” : Submit : Submit_Read [ navigationFlowTo uiCt_InvoiceReader ]
]
]

UIContainer uiCt_InvoiceReader : Window [
component uiCo_ReadInvoice “Consult Invoice” : Form [
dataBinding e_Invoice

part customer “Customer” : Field : Field_Output
[ dataAttributeBinding e_Customer.customerName ]
part dateCreation “Creation Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.dateCreation ]
part dateApproval “Approval Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.dateApproval ]
part datePaid “Payment Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.datePaid ]
part dateDeleted “Delete Date” : Field : Field_Output
[ dataAttributeBinding e_Invoice.dateDeleted ]
part totalValueWithoutVAT “Total Value Without VAT” : Field : Field_Output
[ dataAttributeBinding e_Invoice.totalValueWithoutVAT ]
part totalValueWithVAT “Total Value With VAT” : Field : Field_Output
[ dataAttributeBinding e_Invoice.totalValueWithVAT ]
]

event ev_cancel “Back” : Submit : Submit_Back [ navigationFlowTo Invoices ]
]
```

Spec. 2. ASL specification of a Form and a List for the “Invoice” data entity (IMS example).

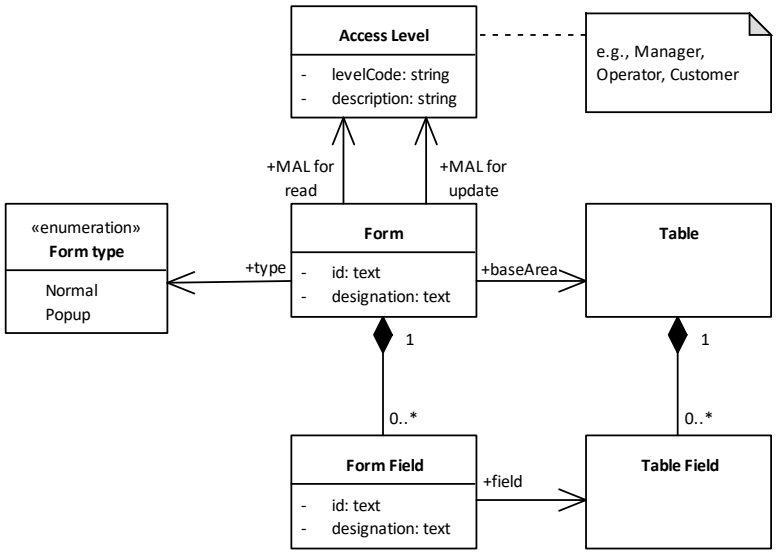

Figure 7. Partial Genio metamodel with form-related constructs (UML notation).

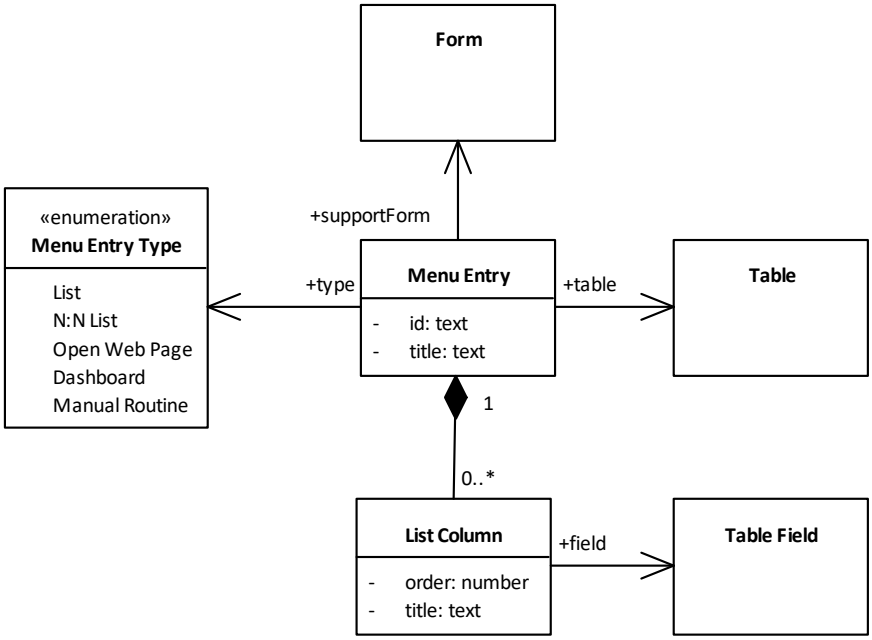

Figure 8. Partial Genio metamodel with list-related constructs (UML notation).

In Genio, a support form is a form that is used to perform CRUD operations of a given table in the context of a specific list. As shown in Figure 8, Genio's user interface makes it easy to specify which form should be used for this purpose. This is possible since, during code generation, Genio automatically creates the appropriate buttons in the user interface to support this behaviour. We can specify the necessary buttons and their proper actions to achieve similar ASL results.

3.3. Actors and Use Cases

ASL supports the specification of actors and use cases. Actors represent the entities that interact with the system, namely user roles and external systems [31,34]. Use cases represent a sequence of actions performed by the system's actors, such as CRUD operations, as well as other specific actions (e.g., approve, export) [31,34]. Figure 9 shows a fragment of ASL's metamodel with a focus on actors and use cases. For instance, it shows that: (1) UseCase aggregates a set of UCActions, operates around a given DataEntity, and has a primary Actor; (2) the primary Actor of the UseCase can either be a user of the system or an external system; (3) a UseCase can be classified as a certain UseCaseType and each UCAction as an ActionType.

Spec. 3 shows a partial ASL specification of actors (e.g., aU_Operator and aU_Customer) and use cases (e.g., uc_CreateInvoice and uc_PrintInvoice). For instance, the use case uc_CreateInvoice defines the actor aU_Operator, who initiates it, and the involved data entity e_Invoice.

As defined in ASL, the concept of actor partially exists in Genio, but only the equivalent of ASL's actors of type "User" can be defined. The concept of use case is supported in ASL but not in Genio. Instead, Genio supports the definition of access levels and access rights. Access levels can be assigned to the end-users (e.g., "Manager", "Customer"), while access rights define types of CRUD operations on data entities. For example, users with the access level "Customer" may read or consult invoices, and only users with the access level "Manager" may update invoices. Use cases are not explicitly supported in Genio but are an implicit aspect resulting from the combination of access levels with access rights, as shown in Figure 7.

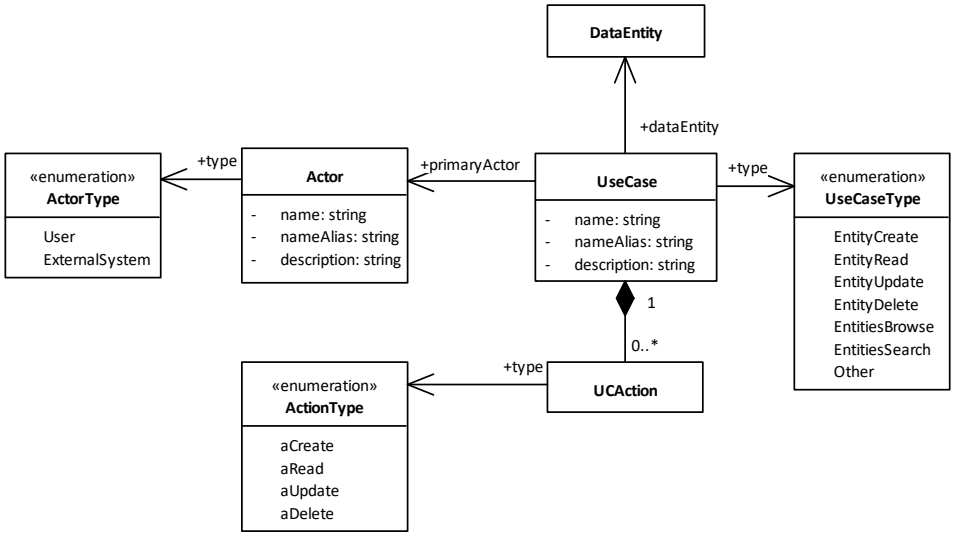

Figure 9. Partial ASL metamodel focused on actors and use cases (UML notation).

```
Actor aU_Operator "Operator": User [ description "Operator manages Invoices and Customers"]
Actor aU_Customer "Customer" : User [ description "Customer receives Invoices to pay" ]

UseCase uc_CreateInvoice "Create Invoice" : EntityCreate [
actorInitiates aU_Operator
dataEntity e_Invoice
actions aCreate
]
UseCase uc_PrintInvoice "Print Invoice" : EntityReport [
actorInitiates aU_Customer
dataEntity e_Invoice
actions aRead
]
```

Spec. 3. Example specification of actors and use cases in ASL.

4. ASL Extensions

This research identified concepts supported by ASL and Genio languages, which our transformation tool can convert directly. However, the study also identified some concepts that ASL did not include at the beginning of this work. This section presents the extensions added to ASL to improve its alignment with Genio's metamodel and its flexibility as a platform-independent specification language. Figure 10 shows the extended ASL metamodel focusing on the proposed changes.

4.1. Data Attribute's Formulas

The first extension added to ASL was the support for specifying formulas defined at the data attribute level. Formulas are expressions that are evaluated at different moments of the application's runtime (e.g., when a record is saved in the database) to compute a value of a corresponding data attribute (or field) based on the value(s) of other data attribute(s).

As shown in Figure 4, Genio supports the specification of different types of formulas. Upon analysis, we proposed two types of formulas: arithmetic and details formulas.

Arithmetic formula. An arithmetic formula supports the specification of basic arithmetic operations (e.g., addition, subtraction, multiplication, or division) to compute the value of its associated data attribute. For instance, considering a data attribute "Value With VAT" of the data entity "InvoiceLine", an arithmetic formula defined by "formula arithmetic (e_InvoiceLine.valueWithoutVAT * e_Product.VATValue)" can be used to calculate the value (including VAT) of the invoice line.

Details formula. A details formula allows the specification of operations on related data entities of a given data entity. For example, considering a data attribute "Total

lines" of the data entity "Invoice", a details formula defined by "formula details: count (e_InvoiceLine)" can be used to calculate the total number of lines of an invoice.

To implement the rigorous definition of such formulas, we add a simple yet flexible expression language [19]. The syntax of this language is similar to other expression languages, such as the one in Microsoft Excel [40]. Spec. 4 shows an example usage of this expression language to specify both arithmetic and details formulas in an improved specification of the "Invoice" data entity.

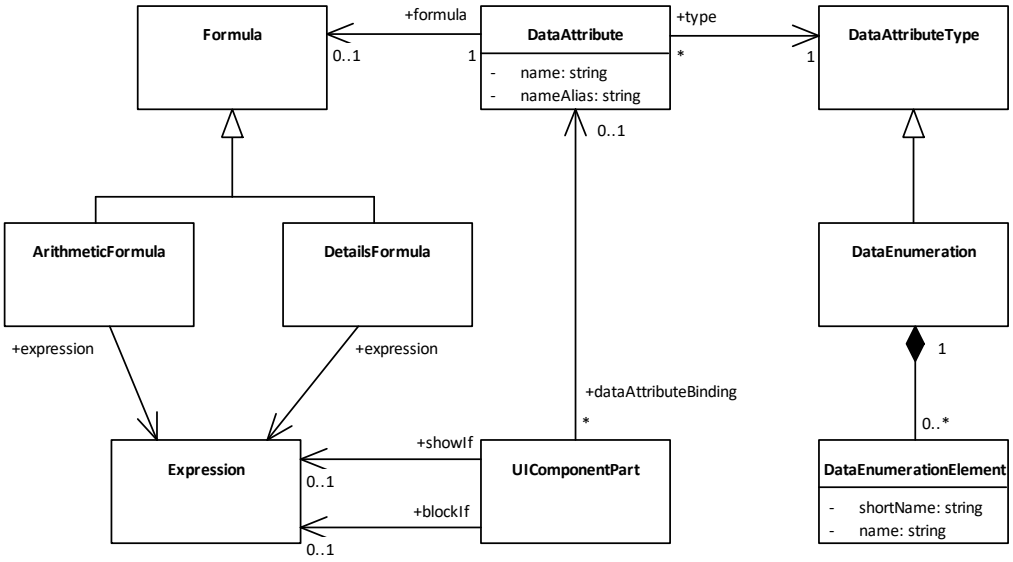

Figure 10. Partial ASL metamodel with the proposed extensions (UML notation).

```
DataEntity e_Invoice "Invoice" : Document [
attribute ID "Invoice ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute customerID "Customer ID" : Integer [ constraints (NotNull ForeignKey(e_Customer)) ]
attribute dateCreation "Creation Date" : Date [defaultValue "today" constraints (NotNull)]
attribute dateApproval "Approval Date": Date
attribute datePaid "Payment Date" : Date
attribute dateDeleted "Delete Date" : Date
attribute isApproved "Is Approved" : Boolean [defaultValue "False"]
attribute totalValueWithoutVAT "Total Value Without VAT" : Decimal(16.2) [
formula details : sum (e_InvoiceLine.valueWithoutVAT)
constraints (NotNull)
]
attribute totalValueWithVAT "Total Value With VAT" : Decimal(16.2) [
formula details : sum (e_InvoiceLine.valueWithVAT)
constraints (NotNull)
]
attribute totalInvoiceLines "Total invoice lines": Integer [
formula details : count (e_InvoiceLine)
]
]

DataEntity e_InvoiceLine "InvoiceLine" : Document [
attribute ID "InvoiceLine ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute invoiceID "Invoice ID" : Integer  [constraints (NotNull ForeignKey(e_Invoice))]
attribute productID "Product ID" : Integer [constraints (NotNull ForeignKey(e_Product))]
attribute order "InvoiceLine Order" : Integer [constraints (NotNull)]
attribute valueWithoutVAT "Value Without VAT" : Decimal
attribute valueWithVAT "Value With VAT" : Decimal [
formula arithmetic (e_InvoiceLine.valueWithoutVAT * e_Product.VATValue)
]
]
```

Spec. 4. Example usage of formulas for the ASL specification of the "Invoice" data entity.

4.2. UI Elements' Expressions

Using the same general purpose expression language initially added to ASL to support the specification of formulas at the data attribute level, it is possible to define other types of expressions, such as conditions that can help shape the user interface. To this end, we

propose two additional usages of expressions at the UIComponentPart level: ShowIf and BlockIf conditions.

ShowIf condition. A ShowIf condition is an expression associated with a UIComponentPart that determines whether it should be displayed in the user interface or not. For instance, considering a Form field "Customer Name" with the ShowIf condition defined by "showIf (e_Invoice.customerID !="")", the ShowIf condition specifies that the field shall be hidden if the value of the data attribute e_Invoice.customerID is empty.

BlockIf condition. A BlockIf condition is similar to a ShowIf condition as both are expressions specified at the UIComponentPart level to manipulate the user interface. Unlike a ShowIf condition, a BlockIf condition determines whether the part should be visible but blocked from accepting user input rather than being completely removed from the user interface. For example, considering a Form field "Customer Name" with the BlockIf condition defined by "blockIf (e_Invoice.customerID !="")", the BlockIf condition specifies that the field shall be blocked if the value of the data attribute e_Invoice.customerID is empty.

4.3. Extended Data Enumerations

Another extension added to ASL is the support of key/value pairs of strings to define data enumerations. Previously, ASL only supported the specification of data enumerations with values (without an associated unique identifier). As discussed in Section 3.1, using this unique identifier for the data enumeration element brings some advantages. For example, it makes it possible to uniquely identify an element in multi-language systems where the associated value is translated according to the user's preferred language. Additionally, it contributes to keeping the size of databases as low as possible since the *keys* are potentially significantly smaller than their associated *values*.

Spec. 5 shows an example specification of the data enumeration "Document Type", using the newly added unique identifiers.

```
1 DataEnumeration DocumentType "Document Type" values (
2 SI "Standard Invoice",
3 CI "Credit Invoice",
4 DI "Debit Invoice",
5 MI "Mixed Invoice"
6 )
```

Spec. 5. Example specification of the data enumeration "Document Type".

5. ASL2Genio Transformation

The ASL2Genio transformation is another extension to ASL (namely, to ASL's code generator), which makes it available in any software tool offered within the ITLingo ecosystem that supports ASL specifications. For Eclipse-based development environments (ITLingo-Studio) [41], the artefacts are continuously generated as the user changes the source specification file. These artefacts are available locally in the user's Eclipse workspace. Complementary and for Web/Cloud-based editors (ITLingo-Cloud), the user may click a generation button to start this task when needed. In that case, the server generates the artefacts and packages them as a zip file so the user can download and use them later in the scope of the "Import Genio Project" task.

Since Genio's import and export mechanism is designed to represent Genio projects in XML format and with a specific directory structure, the proposed generator must meet these requirements. For instance, some concepts, such as data enumerations, must be generated within the same XML file, while others, such as data entities, are expected to be placed in separate files (one file per data entity). Table 1 overviews the expected file and folder structure of the generated artefacts.

As mentioned above, the ASL2Genio transformation extends to ASL's code generator. This code generator is built using the Xtend framework [42] and handles the initial parsing of an ASL specification file. Then, based on the requested target platform (currently, the Genio platform), the generator instantiates the appropriate extension.

Figure 11 shows the involved tasks to transform an ASL specification into a Genio project, including the ASL2Genio transformation and the creation of the Genio project using the generated XML files.

Table 1. Overview of the structure of Genio XML files.

ASL Element	Genio Element	Folder	File Name (.xml)
DataEntities	Tables	GENTABEL	{Data entity name}
UIComponent	Forms	GENFORMS	{Form name}
UIComponent	Menus	GENMENUS	{Menu name}
-	Modules	GENMODUL	{Module name}
DataEnumerations	Enumerations	-	GENARRS
Actors + UseCases	Access levels	-	GENNIVAC

As discussed in Section 1, this research focuses on designing and implementing the ASL2Genio transformation. Spec. 6 shows the pseudo-code of the generate method of ASL's Genio extension, which was implemented based on the aspects discussed in Section 3 and according to the structure outlined in Table 1.

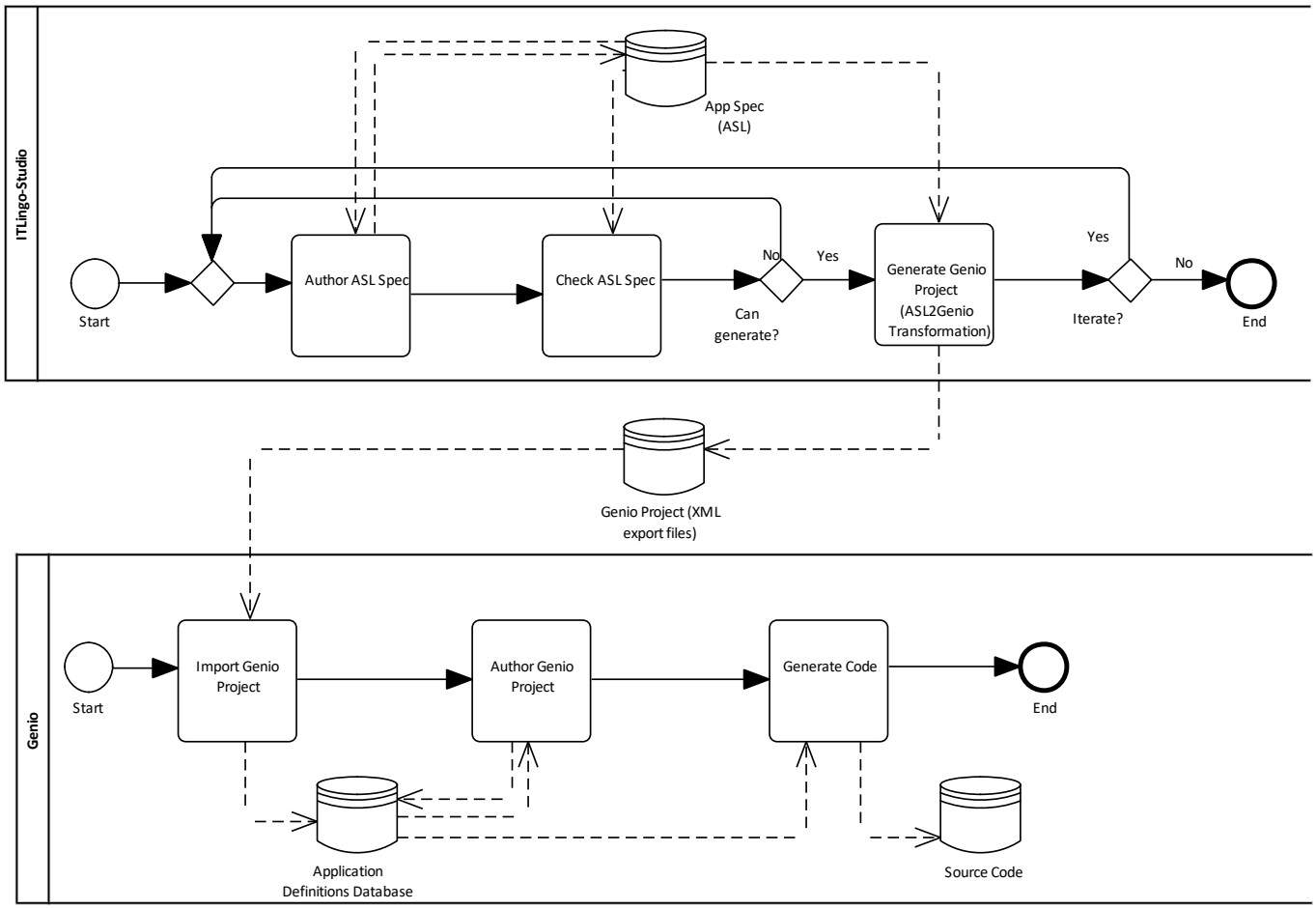

Figure 11. Proposed approach: Combining ASL with the Genio platform (BPMN notation).

```
def void generate() {
// Create a new Genio project
this.createNewProject()

//Normalises identifiers to accommodate Genio constraints
this.normalizeIdentifiers()

// Generate data enumerations (target: GENARRS.xml)
fileSystem.generateFile("/GENARRS.xml", this.getEnumerations().compile())

// Generate access levels (NIVAC and NIVMO) (target: GENNIVAC.xml)
fileSystem.generateFile("/GENNIVAC.xml", this.getAccessLevels().compile())

// Generate data entities (target: GENTABEL/*.xml)
for (tabel : this.getTables()) {
fileSystem.generateFile("/GENTABEL/" + table.name + ".xml", table.compile())
}

// Generate forms (target: GENFORMS/*.xml)
for (form : this.getForms()) {
fileSystem.generateFile("/GENFORMS/" + form.name + ".xml", form.compile())
}

// Generate default module (target: GENMODUL/*.xml)
var module = this.getDefaultModule()
fileSystem.generateFile("/GENMODUL/" + module.Codiprog + ".xml", module.compile())

// Generate menu entries (target: GENMENU/*.xml)
for (menu : this.getMenus()) {
fileSystem.generateFile("/GENMENUS/" + menu.name + ".xml", menu.compile())
}
}
```

Spec. 6. Pseudo-code of the generate method of ASL's Genio extension.

6. Case Study: Invoice Management System

This section presents a case study based on the fictitious "Invoice Management System" (IMS). A summary of the informal requirements of IMS is introduced in Section 3. This case study refers to the specification and development of the IMS application based on the proposed approach.

6.1. ASL Specification

The process of transforming rigorous requirements into a software business application starts with their specification. The entire ASL specification of the Invoice Management System can be consulted in [43]. However, Appendix A presents key aspects that define the IMS application based on the following views: data (data entities and enumerations), user interface (UI elements), and use case view (actors and use cases elements).

6.2. Genio Model

Taking the ASL specification as input, we use the ASL2Genio generator to create the Genio model in XML. Then, we use Genio to import the XML files and create a new project. Figure 12 shows the table "Invoice" definition, initially specified in ASL. Similarly, the transformation engine converted any other ASL concepts mapped to Genio, such as data enumerations, forms, menus, and roles.

The correctness of the transformation rules can be extensively checked by the vast number of validations that Genio performs on project definitions. These validations check the specifications for inconsistencies, such as invalid relationships between tables, helping

to reveal any problems in the ASL2Genio transformation. However, defining a set of unit tests in ASL could provide more assurance of the reliability of these transformations.

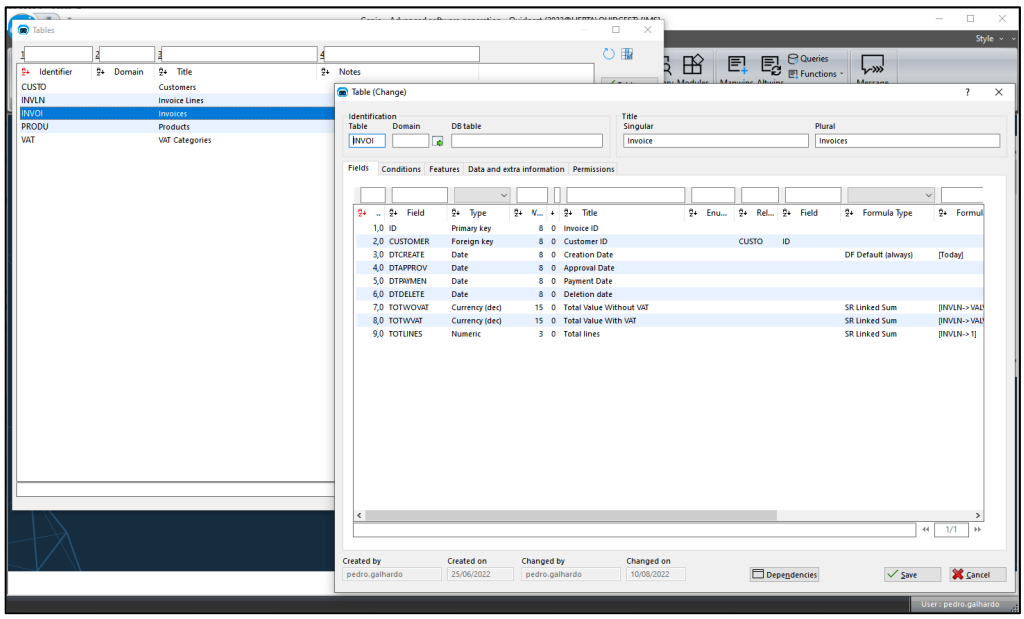

Figure 12. Definition of table "Invoice" in Genio.

6.3. Software Business Application

To conclude the process of transforming a rigorous requirements specification into a software business application, we use Genio's code generation capabilities. Genio supports source code generation to different target platforms and technologies, such as Backoffice C++, ASP.NET MVC, or REST Webservices [8]. Genio's default generation target for Web Applications is ASP.NET MVC [44]. Figure 13 illustrates the process of generating the source code for the project.

Once the code generation task is finished, the steps to compile and deploy the application are the same as if the code had been written manually, i.e., as in traditional approaches. Figure 14 shows a menu page of the Invoice Management System—the list of registered products.

The user may also perform CRUD operations on the invoices, supported by the "Invoice" form, as shown in Figure 15.

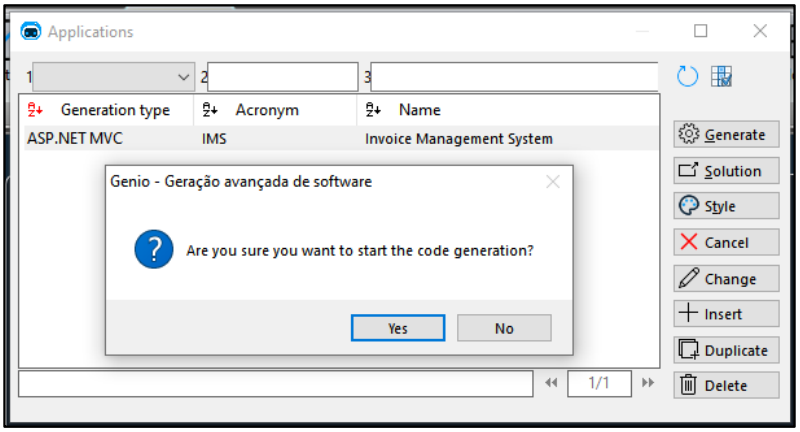

Figure 13. Generating the code for the IMS project in Genio.

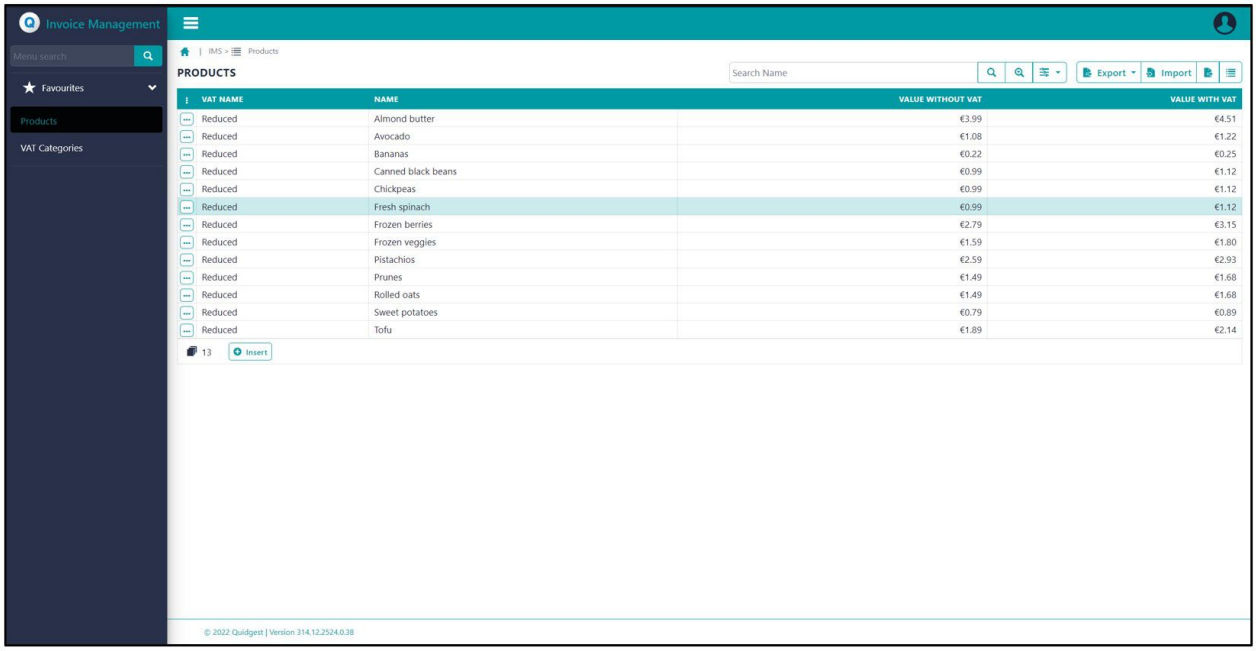

Figure 14. List of registered products in the Invoice Management System.

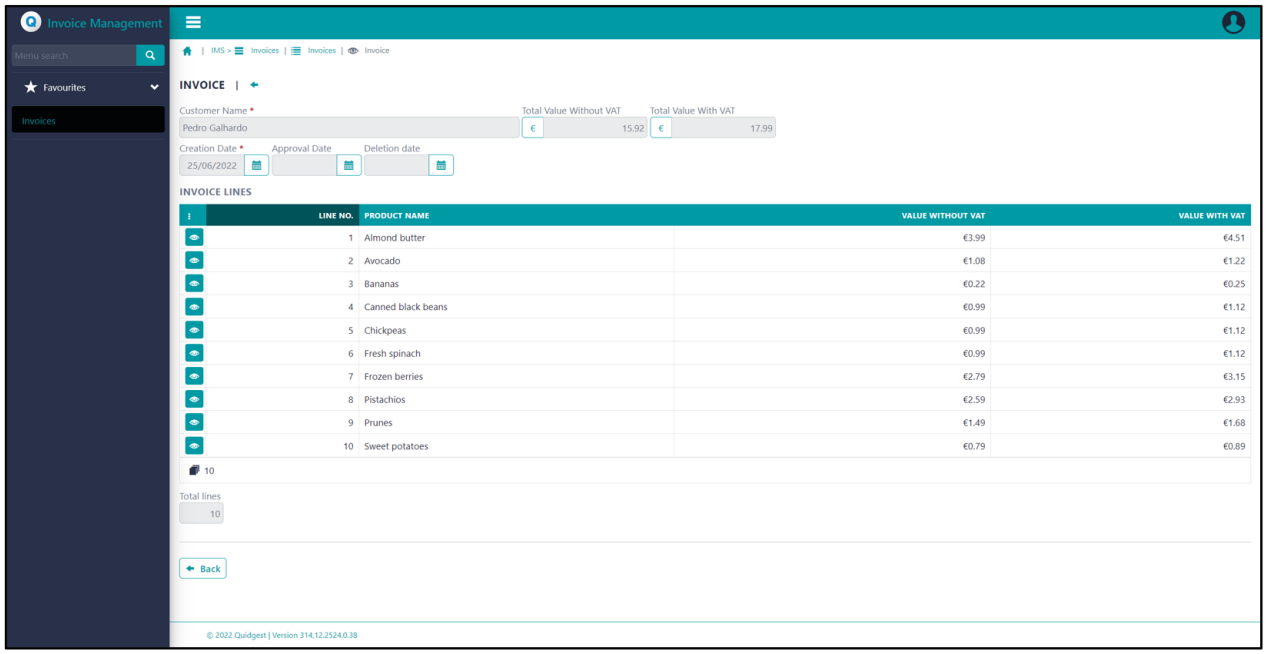

Figure 15. Invoice form (with attributes and invoice lines).

7. Related Work

The recent interest in using low-code platforms to develop software applications has attracted attention from both the industry and the scientific community. Some studies have been conducted to work out common limitations expressed by customers of almost every low-code platform and opportunities for combining model-driven with low-code development approaches. This section presents the related work involving the following aspects: languages and architectural views, model-driven tool interoperability, and low-code platforms' interoperability.

7.1. Languages and Architectural Views

IT organisations have used system and software architecture descriptions to improve communication among stakeholders and enable them to work more comprehensively and consistently [45,46]. The ISO/IEC/IEEE 42010:2011 standard defines software engineering architecture description based on the following concepts [47]: the "architecture" encompasses key ideas or characteristics of a system included in its parts, relationships, and principles of its design and development; the "architecture view" shows the architecture of a system from a specific perspective; and the "architecture description" is a work product used to express an architecture.

In the scope of software development, Kruchten discusses software architecture blueprints based on the "4+1 view model" that encompasses the following architectural views based on the UML usage: scenarios (or use case), logical, development, process, and physical views [48]. More recently, Górski proposes the "1+5 architectural views model" (also based on UML and UML extensions) for the design of cooperating information systems and especially blockchain solutions [49].

However, based on the ASL and Genio languages, the architectural views discussed in our paper are just focused on data, user interface, and use case views. The "data view" and "use case view" are relatively aligned with, respectively, the "logical view" and the "scenarios view" as proposed by the "4+1 view model" and "1+5 view model". On the other hand, these architecture models (as well as modelling languages like UML or SysML) do not include the "user interface view" as discussed in this paper, as well as found in languages and MDE tools like WebRatio [50], XIS-Mobile [51], XIS-Web [52], Enterprise WAE [53], or Kroki [54].

The design of ASL language was strongly influenced by the architecture of modelling languages like XIS-Mobile and XIS-Web (despite being defined as UML profiles while ASL is a textual controlled-natural language). However, none of these approaches has proposed to combine its MDE features with low-code platforms as proposed and discussed in this research.

7.2. Model-Driven Tool Interoperability

Several pieces of research have addressed interoperability in software engineering since the 1980s [55]. Depending on the problem addressed, different approaches have been proposed, operating at distinct levels that achieve different degrees of interoperability. We analyse existing research on how tool interoperability can be achieved using model-driven approaches [14].

Bézivin et al. discuss how model-driven engineering approaches can be employed to solve practical engineering problems [56]. Much like ours, their approach to this challenge is to use small DSLs with well-focused metamodels, rather than large, generic modelling languages like UML 2.0. In their work, the authors propose a tool that makes it possible to perform model transformations to convert the specifications of one tool to another. Additionally, they identify a series of necessary steps to develop such a tool. This information, along with the general lessons and conclusions outlined towards the end of the paper, is valuable to our work.

Markus Voelter and Eelco Visser investigate using domain-specific languages to represent variability [57]. The authors believe that DSLs can be used to bridge the gap observed between current and future models and programming languages. A very similar problem is observed between different low-code platforms, which represents the primary motivation of our work. Similarly, we employ a DSL to address this problem.

Brunelière et al. discuss that approaches that operate at the API level are often too limited to achieve real data interoperability [58]. This issue is the primary motivation for their work, which proposes the construction of a metamodel-level bridge, similar to the one we suggest, to achieve interoperability between tools with variable metamodels. The authors identify a series of four main steps involved in the creation of such a bridge between distinct metamodels: (1) transcription, (2) syntactic translation, (3) semantic alignment, and (4) data interchange.

7.3. Low-Code Platforms' Interoperability

The industry and scientific community's interest in using low-code platforms to develop software applications has attracted attention. Consequently, several studies have been conducted to identify or solve common limitations expressed by customers and practitioners.

To our knowledge, no studies have addressed low-code platforms' interoperability with concrete solutions. However, we identified some studies that use DSL approaches to address certain limitations of low-code platforms.

Bragança et al. discuss how SPL engineering can be supported in low-code platforms using a DSL approach [59]. The proposed solution involves a model-to-model transformation, which takes as input a representation of the low-code model, such as a JSON file, and generates its corresponding low-code metamodel. However, the authors classify this task as semi-automatic; an expert on the low-code platform verifies the generated metamodel. In our work, the low-code metamodel is not automatically generated. We propose a tool that has prior knowledge of the low-code platforms' metamodels that it supports. Naturally, this approach requires active maintenance to support the latest versions of each low-code platform, but it allows the transformation process to be completely automatic.

8. Conclusions

This article proposes an end-to-end model-driven approach that aims to accelerate the development of software business applications by combining rigorous specifications with common features provided by emerging low-code platforms. In particular, this paper discusses a concrete application of this approach based on the ASL language and the Genio platform. Moreover, it discusses the model-to-model transformation, i.e., from ASL specifications into Genio projects (the ASL2Genio transformation). It then leverages the code generation capabilities of Genio to generate source code for concrete software applications (the Genio2Code transformation).

The experience of applying this approach (with the supported tools) to write and transform the Invoice Management System's ASL specification into a Genio project allowed us to identify some benefits and limitations. A key benefit of this approach is that it allows developers to write rigorous specifications in a platform-independent language without sacrificing the code generation capabilities of low-code platforms. As the tool gains support for more low-code platforms, developers can test their code generation capabilities without necessarily learning them first. Additionally, because ASL specifications are platform-independent, this could promote interoperability between low-code applications.

On the other hand, a limitation of this approach could be the difficulty of maintaining the solution. Currently, the transformation mechanism only supports the transformation of ASL specifications into Genio projects, which could be challenging to maintain. As other low-code platforms would be supported, this could become complex to manage. Despite this issue, the proposed approach can also increase the quality of requirements specifications and accelerate the development of business applications by digital-savvy citizen developers.

For future work, we aim to explore the combination of ASL with other low-code platforms and research their reverse transformations (in this case, from Genio projects into ASL specifications). If paired, these two developments could represent a promising step towards achieving interoperability between low-code platforms and, consequently, a higher level of independence from vendor-specific solutions. Following the work de-

veloped recently [32,37,60], we also intend to research linguistics patterns and practical guidelines to better specify business applications, in particular, those most related to the user interface aspects. Furthermore, we plan to research intra- and inter-dependencies between ASL constructs and related languages [61,62], considering the minimisation of their combinatorial effects [63]. Finally, we intend to explore additional concepts and transformations to increase the overall quality and productivity of the proposed approach, for instance, considering emerging areas of blockchain and smart contracts [49,64], robotic process automation [62,65,66], and hyperautomation applications [67].

Author Contributions: Conceptualization, P.G. and A.R.d.S.; Investigation, P.G.; Writing—original draft, P.G.; Writing—review & editing, A.R.d.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. ASL Specification of the IMS Application

This appendix includes the ASL specification for the IMS (Invoice Management System) application used throughout this paper to support the explanation and discussion of the proposed approach.

```
/* ASL specification of the Invoice Management System (IMS) */

Package ims

/*******************************************************
System definition
*******************************************************/

System IMS "Invoice Management System": Application : Application_Web

/*******************************************************
Data view: Data Entities
*******************************************************/

DataEntity e_VAT "VAT Category" : Reference [
attribute VATCode "VAT code" : String [constraints (PrimaryKey NotNull Unique)]
attribute VATName "VAT name": String
attribute VATValue "VAT value": String
]

DataEntity e_Product "Product" : Master [
attribute ID "Product ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute vatCode "VAT code" : Integer [ constraints (NotNull ForeignKey(e_VAT)) ]
attribute productName "Name" : String(100)
attribute valueWithoutVAT "Value Without VAT" : Decimal(16.2) [constraints (NotNull)]
attribute valueWithVAT "Value With VAT" : Decimal(16.2) [constraints (NotNull)]
attribute VATValue "VAT value" : Decimal [ formula arithmetic (e_VAT.VATValue) ]
]

DataEntity e_Customer "Customer" : Master [
attribute ID "Customer ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute customerName "Name" : String(100)
attribute fiscalID "Fiscal ID" : String(9)
attribute logoImage "Logo image" : Image
attribute address "Address" : String(200)
attribute IBAN "IBAN": String(34)
attribute SWIFT "SWIFT" : String(8)
]

DataEntity e_CustomerVIP "CustomerVIP" : Master [
attribute ID "CustomerVIP ID" : Integer [constraints (PrimaryKey NotNull Unique)]
attribute customerID "Customer ID" : Integer [ constraints (NotNull ForeignKey(e_Customer)) ]
```

```
attribute discountRate “Discount rate” : Integer
]

DataEntity e_Invoice “Invoice” : Document [
attribute ID “Invoice ID” : Integer [constraints (PrimaryKey NotNull Unique)]
attribute type “Type” : DataEnumeration enum_DocumentType
attribute customerID “Customer ID” : Integer [ constraints (NotNull ForeignKey(e_Customer)) ]
attribute dateCreation “Creation Date” : Date [defaultValue “today” constraints (NotNull)]
attribute dateApproval “Approval Date”: Date
attribute datePaid “Payment Date” : Date
attribute dateDeleted “Delete Date” : Date
attribute isApproved “Is Approved” : Boolean [defaultValue “False”]
attribute totalValueWithoutVAT “Total Value Without VAT” : Decimal(16.2) [
formula details : sum (e_InvoiceLine.valueWithoutVAT)
constraints (NotNull)
]
attribute totalValueWithVAT “Total Value With VAT” : Decimal(16.2) [
formula details : sum (e_InvoiceLine.valueWithVAT)
constraints (NotNull)
]
attribute totalInvoiceLines “Total invoice lines”: Integer [
formula details : count (e_InvoiceLine)
]
]

DataEntity e_InvoiceLine “InvoiceLine” : Document [
attribute ID “InvoiceLine ID” : Integer [constraints (PrimaryKey NotNull Unique)]
attribute invoiceID “Invoice ID” : Integer    [constraints (NotNull ForeignKey(e_Invoice))]
attribute productID “Product ID” : Integer [constraints (NotNull ForeignKey(e_Product))]
attribute order “InvoiceLine Order” : Integer [constraints (NotNull)]
attribute valueWithoutVAT “Value Without VAT” : Decimal
attribute valueWithVAT “Value With VAT” : Decimal [
formula arithmetic (e_InvoiceLine.valueWithoutVAT * e_Product.VATValue)
]
]

/*******************************************************
Data view: Data enumerations
*******************************************************/

DataEnumeration enum_DocumentType “Document Type” values (
SI “Standard Invoice”,
CI “Credit Invoice”,
DI “Debit Invoice”,
MI “Mixed Invoice”
)

/*******************************************************
User interface view: Forms
*******************************************************/

// “Invoice” Form

component uiCo_InvoiceForm “Invoice” : Form [
dataBinding e_Invoice

part customer “Customer” : Field : Field_Input
[ dataAttributeBinding e_Customer.customerName ]
part dateCreation “Creation Date” : Field : Field_Input
[ dataAttributeBinding e_Invoice.dateCreation ]
part dateApproval “Approval Date” : Field : Field_Input
[ showIf (e_Invoice.dateCreation != ““) dataAttributeBinding e_Invoice.dateApproval ]
part datePaid “Payment Date” : Field : Field_Input
[ showIf (e_Invoice.dateApproval != ““) dataAttributeBinding e_Invoice.datePaid ]
part dateDeleted “Delete Date” : Field : Field_Input
[ dataAttributeBinding e_Invoice.dateDeleted ]
part totalValueWithoutVAT “Total Value Without VAT” : Field : Field_Input
[ dataAttributeBinding e_Invoice.totalValueWithoutVAT ]
part totalValueWithVAT “Total Value With VAT” : Field : Field_Input
[ dataAttributeBinding e_Invoice.totalValueWithVAT ]
]

UIContainer uiCt_InvoiceCreator : Window [
component uiCo_InvoiceForm

event ev_cancel “Back”: Submit: Submit_Back [navigationFlowTo Invoices]
event ev_save “Save”: Submit: Submit_Create [navigationFlowTo Invoices]
]

UIContainer uiCt_InvoiceReader : Window [
component uiCo_InvoiceForm

event ev_cancel “Back” : Submit : Submit_Back [ navigationFlowTo Invoices ]
]

UIContainer uiCt_InvoiceEditor : Window [
```

```
component uiCo_InvoiceForm

event ev_cancel "Back" : Submit : Submit_Back [ navigationFlowTo Invoices ]
event ev_save "Save" : Submit : Submit_Update [ navigationFlowTo Invoices ]
]

// "Customer" Form

component uiCo_CustomerForm "Customer" : Form [
dataBinding e_Customer

part customerName "Name" : Field
[ dataAttributeBinding e_Customer.customerName ]
part fiscalID "Fiscal ID" : Field
[ dataAttributeBinding e_Customer.fiscalID ]
part logoImage "Logo Image" : Field
[ dataAttributeBinding e_Customer.logoImage ]
part address "Address": Field
[ dataAttributeBinding e_Customer.address ]
part IBAN "IBAN": Field
[ dataAttributeBinding e_Customer.IBAN ]
part SWIFT "SWIFT" : Field
[ dataAttributeBinding e_Customer.SWIFT ]
]

UIContainer uiCt_CustomerCreator : Window [
component uiCo_CustomerForm
event ev_cancel "Back" : Submit : Submit_Back [ navigationFlowTo Customers ]
event ev_save "Save" : Submit : Submit_Create [ navigationFlowTo Customers ]
]

UIContainer uiCt_CustomerReader : Window [
component uiCo_CustomerForm
event ev_cancel "Back" : Submit : Submit_Back [ navigationFlowTo Customers ]
]

UIContainer uiCt_CustomerEditor : Window [
component uiCo_CustomerForm
event ev_cancel "Back" : Submit : Submit_Back [ navigationFlowTo Invoices ]
event ev_save "Save" : Submit : Submit_Update [ navigationFlowTo Invoices ]
]


/*******************************************************
Use case view: Actors & Use Cases
*******************************************************/

Actor aU_TechnicalAdmin "TechnicalAdmin" : User [ description "Admin manage Users, VAT, etc." ]
Actor aU_Operator "Operator": User [ description "Operator manages Invoices and Customers"]
Actor aU_Manager "Manager": User [ description "Manager approves Invoices, etc."]
Actor aU_Customer "Customer" : User [ description "Customer receives Invoices to pay" ]
Actor aS_ERP "ERP" : ExternalSystem [ description "ERP receives info of paid invoices" ]

// TechnicalAdmin

UseCase uc_Manage_Products "Manage Products" : EntitiesManage [
    actorInitiates aU_TechnicalAdmin
    dataEntity e_Product
    actions aCreate, aRead, aUpdate, aDelete
]

UseCase uc_Manage_VAT_Categories "Manage VAT Categories" : EntitiesManage [
    actorInitiates aU_TechnicalAdmin
    dataEntity e_VAT
    actions aCreate, aRead, aUpdate, aDelete
]

// Operator

UseCase uc_Manage_Customers "Manage Customers" : EntitiesManage [
    actorInitiates aU_Operator
    dataEntity e_Customer
    actions aCreate, aRead, aUpdate, aDelete
]

UseCase uc_Manage_Invoices "Manage Invoices" : EntitiesManage [
    actorInitiates aU_Operator
    dataEntity e_Invoice
    actions aCreate, aRead, aUpdate, aDelete
]

// Manager

UseCase uc_ConsultInvoicesToApprove "Consult Invoices to approve" : EntitiesBrowse [
    actorInitiates aU_Manager
    dataEntity e_Invoice
```

```
215      actions aRead
216      extensionPoints aApprove
217  ]
218
219  // Customer
220
221  UseCase uc_Consult_MyInvoices "Consult My Invoices" : EntitiesBrowse [
222      actorInitiates aU_Customer
223      dataEntity e_Invoice
224      actions aRead
225      extensionPoints aPay
226  ]
```

References

1. Shah, T.; Patel, S.V. A Review of Requirement Engineering Issues and Challenges in Various Software Development Methods. *Int. J. Comput. Appl.* **2014**, *99*, 36–45. [CrossRef]
2. Al-Fedaghi, S. Developing Web Applications. *Int. J. Softw. Eng. Its Appl.* **2011**, *5*, 57–68.
3. Gamito, I.; Silva, A.R. From Rigorous Requirements and User Interfaces Specifications into Software Business Applications. In Proceedings of the 13th International Conference on the Quality of Information and Communications Technology (QUATIC'2020), Braga, Portugal, 8–11 September 2020.
4. Bock, A.C.; Frank, U. Low-code platform. *Bus. Inf. Syst. Eng.* **2021**, *63*, 733–740. [CrossRef]
5. Frank, U.; Maier, P.; Bock, A. *Low Code Platforms: Promises, Concepts and Prospects: A Comparative Study of Ten Systems*; ICB-Research Report No. 70; Universität Duisburg-Essen, Institut für Informatik und Wirtschaftsinformatik (ICB): Essen, Germany, 2021. [CrossRef]
6. Overeem, M. Evolution of Low-Code Platforms. Ph.D. Thesis, Utrecht University, Utrecht, The Netherlands, 2022.
7. Di Ruscio, D.; Kolovos, D.; de Lara, J.; Pierantonio, A.; Tisi, M.; Wimmer, M. Low-code development and model-driven engineering: Two sides of the same coin? *Softw. Syst. Model.* **2022**, *21*, 437–446. [CrossRef]
8. PLATFORM—Genio by Quidgest. Available online: https://genio.quidgest.com/platform (accessed on 13 October 2021).
9. OutSystems Evaluation Guide. Available online: https://www.outsystems.com/evaluation-guide (accessed on 13 October 2021).
10. Mendix Evaluation Guide. Available online: https://www.mendix.com/evaluation-guide (accessed on 13 October 2021).
11. Business Apps | Microsoft Power Apps. Available online: https://powerapps.microsoft.com (accessed on 22 August 2022).
12. Google AppSheet | Build Apps with No Code. Available online: https://www.appsheet.com (accessed on 22 August 2022).
13. Build a Better Way to Work | Amazon Honeycode. Available online: https://www.honeycode.aws (accessed on 22 August 2022).
14. Silva, A.R. Model-driven engineering: A survey supported by the unified conceptual model. *Comput. Lang. Syst. Struct.* **2015**, *43*, 139–155.
15. Cabot, J. Positioning of the low-code movement within the field of model-driven engineering. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS' 20), New York, NY, USA, 16–23 October 2020.
16. PROJECTS—Genio by Quidgest. Available online: https://genio.quidgest.com/projects-2 (accessed on 8 July 2022).
17. Silva, A.R. ITLingo Research Initiative in 2022. *arXiv* **2022**, arXiv:1804.00344.
18. Deursen, A.V.; Klint, P.; Visser, J. Domain-specific languages. *ACM Sigplan Not.* **2000**, *35*, 26–36. [CrossRef]
19. Kosar, T.; Oliveira, N.; Mernik, M.; Pereira, V.J.M.; Črepinšek, M.; Da, C.D.; Henriques, R.P. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. *Comput. Sci. Inf. Syst.* **2010**, *438*, 247–264. [CrossRef]
20. HTML: HyperText Markup Language. Available online: https://developer.mozilla.org/en-US/docs/Web/HTML (accessed on 20 June 2022).
21. CSS: Cascading Style Sheets. Available online: https://developer.mozilla.org/en-US/docs/Web/CSS (accessed on 20 June 2022).
22. LaTeX—A Document Preparation System. Available online: https://www.latex-project.org (accessed on 20 June 2022).
23. Kurtev, I.; Bézivin, J.; Jouault, F.; Valduriez, P. Model-based DSL frameworks. In Proceedings of the Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications—OOPSLA'06, Portland, OR, USA, 22–26 October 2006.
24. Sanchis, R.; García-Perales, Ó.; Fraile, F.; Poler, R. Low-Code as Enabler of Digital Transformation in Manufacturing Industry. *Appl. Sci.* **2020**, *10*, 12. [CrossRef]
25. How Enterprise-Grade Low-Code Speeds up Time to Market. Available online: https://www.nearpartner.com/2021/02/how-enterprise-grade-low-code-speeds-up-time-to-market (accessed on 20 October 2021).
26. Gartner Says the Majority of Technology Products and Services Will Be Built by Professionals Outside of IT by 2024. Available online: https://www.gartner.com/en/newsroom/press-releases/2021-06-10-gartner-says-the-majority-of-technology-products-and-services-will-be-built-by-professionals-outside-of-it-by-2024 (accessed on 20 October 2021).
27. Danilchenko, Y.B. *Automatic Code Generation Using Artificial Intelligence*; ProQuest/UMI: Ann Arbor, MI, USA, 2012.
28. Event Sourcing. Available online: https://martinfowler.com/eaaDev/EventSourcing.html (accessed on 23 August 2022).
29. What Is API Management? Available online: https://www.redhat.com/en/topics/api/what-is-api-management (accessed on 23 August 2022).

30. Evolutionary Architecture: Supporting Constant Change. Available online: https://www.sqli.nl/en/blog/evolutionary-architecture (accessed on 23 August 2022).
31. Silva, A.R. Rigorous Specification of Use Cases with the RSL Language. In Proceedings of the 28th International Conference on Information Systems Development (ISD'2019), Toulon, France, 28–30 August 2019.
32. Silva, A.R.; Savić, D. Linguistic Patterns and Linguistic Styles for Requirements Specification: Focus on Data Entities. *Appl. Sci.* **2021**, *11*, 4119. [CrossRef]
33. Silva, A.R. Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-level Language. In Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLOP'2017), Irsee, Germany, 12–16 July 2017.
34. Silva, A.R. Linguistic Patterns, Styles, and Guidelines for Writing Requirements Specifications: Focus on Use Cases and Scenarios. *IEEE Access* **2021**, *9*, 143506–143530. [CrossRef]
35. Flow Modeling Language Specification Version 1.0. Available online: https://www.omg.org/spec/IFML/1.0 (accessed on 21 October 2021).
36. GENIO: Xtreme Low-Code Platform. Available online: https://quidgest.com/en/about-quidgest/genio-platform (accessed on 20 October 2021).
37. Ribeiro, A.; Silva, A.; Silva, A.R. Data Modeling and Data Analytics: A Survey from a Big Data Perspective. *J. Softw. Eng. Appl.* **2015**, *8*, 617–634. [CrossRef]
38. Nielsen, J. *Usability Engineering*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1994.
39. O'Hara, J.M.; Fleger, S. *Human-System Interface Design Review Guidelines*; Technical Report; Brookhaven National Lab (BNL): Upton, NY, USA, 2020.
40. Overview of Formulas. Available online: https://support.microsoft.com/en-us/office/overview-of-formulas-34519a4e-1e8d-4f4b-84d4-d642c4f63263 (accessed on 5 July 2022).
41. Eclipse Foundation. Available online: https://www.eclipse.org (accessed on 25 June 2022).
42. Xtend. Available online: https://www.eclipse.org/xtend (accessed on 4 January 2022).
43. Invoice Management System (IMS) Specified in the ITLingo ASL Language. Available online: https://github.com/pgalhardo/ims (accessed on 25 June 2022).
44. ASP. NET MVC Pattern. Available online: https://dotnet.microsoft.com/en-us/apps/aspnet/mvc (accessed on 25 June 2022).
45. Medvidovic, N.; Taylor, R.N. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.* **2000**, *26*, 70–93. [CrossRef]
46. Hasselbring, W. Software Architecture: Past, Present, Future. In *The Essence of Software Engineering*, 1st ed.; Gruhn, V., Striemer, R., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 169–184.
47. *ISO/IEC/IEEE 42010:2011*; Systems and Software Engineering—Architecture Description. ISO: Geneva, Switzerland, 2011.
48. Kruchten, P. The 4 + 1 View Model of Software Architecture. *IEEE Softw.* **1995**, *12*, 42–50. [CrossRef]
49. Górski, T. The 1 + 5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. *Symmetry* **2021**, *13*, 2000. [CrossRef]
50. Brambilla, M.; Fraternali, P. Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience. *Sci. Comput. Program* **2014**, *89*, 71–87. [CrossRef]
51. Ribeiro, A.; da Silva, A.R. Evaluation of XIS-Mobile, a domain specific language for mobile application development. *J. Softw. Eng. Appl.* **2014**, *7*, 906–919. [CrossRef]
52. Seixas, J.; Ribeiro, A.; da Silva, A.R. A Model-Driven Approach for Developing Responsive Web Apps. In Proceedings of the ENASE, Heraklion, Greece, 4–5 May 2019.
53. Cortés, H.; Navarro, A. Enterprise WAE: A Lightweight UML Extension for the Characterisation of the Presentation Tier of Enterprise Applications with MDD-Based Mockup Generation. *Int. J. Softw. Eng. Knowl. Eng.* **2017**, *27*, 1291–1331.
54. Filipović, M.; Vuković, Ž.; Dejanović, I.; Milosavljević, G. Rapid Requirements Elicitation of Enterprise Applications Based on Executable Mockups. *Appl. Sci.* **2021**, *11*, 7684. [CrossRef]
55. Wicks, M.N.; Dewar, R.G. A new research agenda for tool integration. *J. Syst. Softw.* **2007**, *80*, 1569–1585.
56. Bézivin, J.; Bruneliere, H.; Jouault, F.; Kurtev, I. Model Engineering Support for Tool Interoperability. In Proceedings of the Workshop in Software Model Engineering (WiSME'2005)—A MODELS 2005 Satellite Event, Montego Bay, Jamaica, 2–7 October 2005.
57. Voelter, M.; Visser, E. Product Line Engineering Using Domain-Specific Languages. In Proceedings of the 15th International Software Product Line Conference, Munich, Germany, 21–26 August 2011.
58. Brunelière, H.; Cabot, J.; Clasen, C.; Jouault, F.; Bézivin, J. Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In *Modelling Foundations and Applications*; ECMFA; Springer: Berlin/Heidelberg, Germany, 2010.
59. Bragança, A.; Azevedo, I.; Bettencourt, N.; Morais, C.; Teixeira, D.; Caetano, D. Towards supporting SPL engineering in low-code platforms using a DSL approach. In Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Chicago, IL, USA, 17–18 October 2021.
60. Ferreira, A.; Silva, A.R.; Paiva, A.C. Towards the Art of Writing Agile Requirements with User Stories, Acceptance Criteria, and Related Constructs. In Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'2022), Online, 25–26 April 2022.

61. Paiva, A.C.; Maciel, D.; Silva, A.R. From Requirements to Automated Acceptance Tests with the RSL Language. In *Communications in Computer and Information Science*; Springer: Cham, Switzerland, 2020; Volume 1172.
62. Correia, C.M.; Silva, A.R. Platform-Independent Specifications for Robotic Process Automation. In Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD'2022), Online, 6–8 February 2022.
63. Verelst, J.; Silva, A.R.; Mannaert, H.; Ferreira, D.A.; Huysmans, P. Identifying Combinatorial Effects in Requirements Engineering. In Proceedings of the Third Enterprise Engineering Working Conference (EEWC 2013), Luxembourg, 13–14 May 2013.
64. Górski, T.; Bednarski, J. Applying model-driven engineering to distributed ledger deployment. *IEEE Access* **2020**, *8*, 118245–118261. [CrossRef]
65. Ivančić, L.; Suša Vugec, D.; Bosilj Vukšić, V. Robotic Process Automation: Systematic Literature Review. In Proceedings of the 17th International Conference on Business Process Management (BPM 2019), Vienna, Austria, 1–6 September 2019.
66. Enríquez, J.G.; Jiménez-Ramírez, A.; Domínguez-Mayo, F.J.; Garcia-Garcia, J.A. Robotic process automation: A scientific and industrial systematic mapping study. *IEEE Access* **2020**, *8*, 39113–39129. [CrossRef]
67. Haleem, A.; Javaid, M.; Singh, R.P.; Rab, S.; Suman, R. Hyperautomation for the enhancement of automation in industries. *Sens. Int.* **2021**, *2*, 100124. [CrossRef]