On the Automated Generation of UI for Template-based Requirements Specification

Ikram Darif*, Ghizlane El Boussaidi†, and Sègla Kpodjedo
Department of Software and IT Engineering, École de technologie supérieure
Montreal, Canada
Email: *ikram.darif.1@ens.etsmtl.ca, †ghizlane.elboussaidi@etsmtl.ca

***Abstract*—Requirements specification is a critical phase of the software development life cycle where requirements are identified and documented. To mitigate the ambiguity of natural language, templates can be adopted for the semi-formal specification of requirements. Automated specification support is important as it simplifies and expedites the specification process. However, developing the User Interface (UI) for template-based specification is demanding in terms of time and resources. In this paper, we propose a model-driven approach for generating UIs that support template-based requirements specification. We support the generation through mapping rules that link the template metamodel to the UI metamodel. We provide a systematic four-step process for the generation of UI from an input template, which includes preparation, components identification, rendering, and integration. We implemented our approach into our tool MD-RSuT for the automated generation of UI. To evaluate our approach, we compared it to manual UI development and assessed the quality of generated UIs. Our evaluation indicated that the approach provides multiple advantages over manual development, and the generated UIs adhere to UI design principles of structure, simplicity, visibility, feedback, tolerance, and reuse.**

***Index Terms*—Requirements specification, Requirement templates, Automated UI Generation, Model-driven engineering.**

I. Introduction

Requirements are crucial artifacts in software development. They specify the capabilities that must be met by the system to address the client's needs [1]. Requirements are formally documented within the Software Requirements Specification (SRS), a document that gathers all requirements of a software product [2]. Typically, requirements are specified in Natural Language (NL). However, using NL introduces ambiguity and errors within the specification. To mitigate these issues, the use of templates has emerged for the semi-formal specification of requirements. Templates include fixed parts and placeholders that constrain the specification. They enhance the readability, consistency, and clarity of the specified requirements.

Tool support for template-based requirements specification facilitates the specification and reduces the possibility of errors, particularly for systems with a large number of requirements. User Interfaces (UI) play a key role in such tools, as they serve as the primary means of interaction for the user. However, manually developing a UI for specific templates requires time and continuous efforts to maintain the UI in sync with the templates during their evolution. In this context, automating UI generation is valuable as it significantly reduces the time and efforts, while improving consistency across the generated UIs. However, to the best of our knowledge, there are no approaches that support the automated generation of UI for template-based specification.

In this paper, we build on our previous work [3], [4] to propose an approach for the automated generation of UI for template-based requirements specification. In [3], we proposed a model-driven engineering (MDE) approach for specifying and evolving templates. The proposed approach relies on models that represent templates and a Domain Specific Modeling Language (DSML), called UTL, to specify and constrain the template models [4]. In this paper, we further leverage MDE technologies to specify and generate the UI of the templates. Thus our proposed approach relies on: (1) templates which are represented by template models that are instances of a the UTL metamodel, (2) UI which is represented by UI models that are instances of an UI metamodel, and (3) mapping rules that link the UTL metamodel to the UI metamodel enabling the automatic generation of the UI. The process of generating UIs from templates includes four steps: (1) *Preparation*, where the input template is expressed using UTL, (2) *Components identification*, where UI components are identified using predefined mapping rules, (3) *Rendering*, where a UI is generated using the identified components, and (4) *Integration*, where the generated UI is linked to the backend code. This process supports UI generation for requirements templates of all types, beyond the seven templates proposed in our previous work [3]. We implemented our approach into our tool called MD-RSuT [3]. It allows the automated generation of UI for templates created through MD-RSuT.

To evaluate our approach, we assessed the quality of generated UIs against established UI design principles, and we compared it to manual UI development. Our evaluation showed that our approach provides multiple benefits over manual development, and the UIs adhere to the UI design principles: structure, simplicity, visibility, feedback, tolerance, and reuse.

The structure of the paper is organized as follows. In Section II, we review the related work. Section III presents our approach and tool support. Section IV presents the evaluation and the threats to validity. Finally, Section V concludes our paper and presents our future work.

II. Related work

Several approaches were proposed to support the automated generation of UIs. For instance, Kolthoff et al. [5], [6] present

an approach called GUI2R, that semi-automates the generation of graphical UI (GUI) prototypes for mobile applications from unconstrained NL requirements. They use Natural Language Processing (NLP) and Information Retrieval (IR) techniques to process requirements that are mapped to mobile GUI repositories to identify relevant GUI components [5]. Juárez-Ramírez et al. [7] present an approach for the automated generation of UI prototypes from semi-formal Controlled Natural Language (CNL) requirements. In this approach, the requirements analyst specifies requirements, which are then analyzed by the NLARE tool to identify key elements for building use cases [7]. These elements are used to build use case descriptions, from which the interactions necessary for use cases are identified. Finally, these interactions are translated into UI elements and events [7]. In their study [8], Elkoutbi et al. propose an approach for the automated generation of UI prototypes from user scenarios using the Unified Modeling Language (UML). The scenarios are presented in UML collaboration diagrams, and then transformed into UML statechart specifications. The corresponding UIs are then automatically generated from the identified specifications. While these approaches [5]–[8] are relevant for GUI prototyping from requirements, they do not support the specification of requirements.

Other approaches use model-driven technologies for UI generation. For instance, Rosado et al. [9] propose an approach for the automated generation of UI models from use case and domain models. This is supported by defining: (1) a UML-aligned metamodel for domain and use cases, (2) a MOF-based metamodel for UI models, and (3) a set of transformation rules between the metamodels to support the generation [9]. Another example is proposed by Puerta et al. [10], where an environment for model-based UI development from domain models is proposed. The UI generation is supported by a knowledge-based system that applies a set of dialog design and layout rules. While these approaches use model-driven technologies for UI generation, they are not applicable to support the specification of requirements. To the best of our knowledge, there are no approaches for model-based automated generation of UI for template-based requirements specification.

III. A Model-Driven approach for automated UI generation

To support the generation of UI for template-based specification of requirements, we propose the four-step approach defined in Figure 1. Our approach is model-driven, as: (1) the input templates are represented by template models, which are instances of the UTL metamodel, and (2) the generated UIs are represented by UI models, which are instances of a UI metamodel. UTL is a domain specific modeling language that we introduced in our previous work [4]. We support the automated generation of UI by defining mappings between the template metamodel (UTL) and the UI metamodel.

The UIs for using templates include two portions, one that is common between templates, and another that varies between templates. Our approach covers the generation of the second portion of the UI, which differs across templates. Figure 2 presents an example of a template's UI. The common portion (i.e., the top part in the figure) includes: (1) fields for general information about the template (e.g., name and source), (2) a drop-down list to select the template to be used for the specification, (3) a dynamic text field displaying the requirement's text (i.e., the text is automatically built by the tool using the input of the user through the part that varies: the bottom part in Figure 2), and (4) a HELP button to access information about the selected template. The remaining UI components are specific to the selected template. Our approach focuses on the generation of these components.

Our approach consists of four steps: (1) *Preparation*, where the input template is expressed using the UTL's syntax, (2) *Components identification*, where the UI components are identified by applying mapping rules between the *template metamodel* and the *UI metamodel*, (3) *Rendering*, where the identified UI components are used to build the UI, and (4) *Integration*, where the generated UI is linked to the back-end code. In the following, we explain each step of the approach.

A. Preparation

In this step, the template must be expressed using the syntax of UTL. This ensures that the template conforms to the template metamodel, thus enabling its automated analysis. Figure 3 presents examples of two templates: (1) a simple type definition template, which specifies primitive types (e.g., numeric, alphanumeric) and optionally their expected values, and (2) a composite type definition template, which specifies composite types through a list of parameters and their variability (i.e., whether they are fixed or variable). These templates are written according to UTL's syntax [4]: (1) optional elements are enclosed in brackets "[]", like the expected values part in the simple type definition template; (2) elements with one-to-many occurrences are enclosed in parentheses "()+", like the parameter definition part of the composite type definition template; and (3) some placeholders can be defined by enumerations, like the variability placeholder in the composite type definition template. We will use these templates to illustrate the UI generation process.

B. Components Identification

The second step of the approach is *Components identification*. During this step, the components of the UI are identified based on the input template. This identification is achieved by applying mapping rules that link elements of the template metamodel to their corresponding elements (i.e, the UI components that support their specification) within the UI metamodel. Furthermore, this step establishes the relationships (e.g., containment relationships) between the identified UI components. In the following, we will present the template metamodel, the UI metamodel, and the mapping rules.

1) Templates Metamodel: In our work [4], we introduced UTL, a DSML for requirements templates. As part of UTL, we developed a template metamodel that presents the high-level structure of templates. The UTL metamodel is presented in Figure 4. A requirement template includes one or multiple

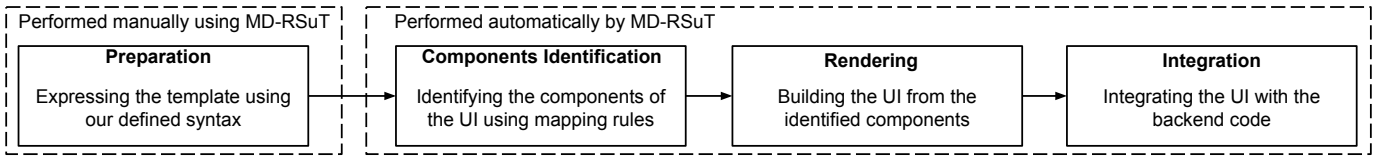


Fig. 1. The UI generation approach

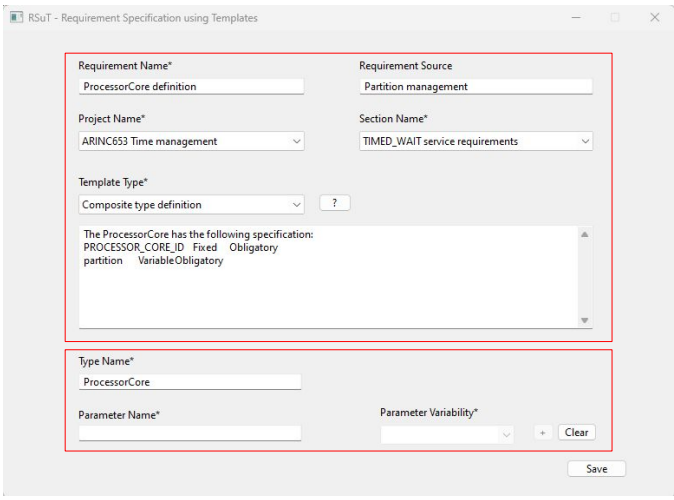


Fig. 2. Specification UI for the composite type definition template

Simple type definition template

The **<type name>** has the following specification:
Type: **<type : Numeric / Alphanumeric / Alphabetic/ Enumeration>**
[Expected values: **<value enumeration>**]

Composite type definition template

The **<type name>** has the following specification:
(**<parameter name>** **<variability: FIXED / VARIABLE>**)+... (one per line)

Fig. 3. Examples of templates

blocks. Each block can contain one to multiple statements, i.e., lines of text. Blocks and statements are defined by their name, their order within the template and block, respectively, and their multiplicity, which indicates the number of instances that can be specified within requirements. Also, a statement is defined by ”isOnePerLine” attribute, which specifies whether instances of a statement should be written each in a separate line. For instance, the simple type definition template (Figure 3) includes two blocks: one with a 1-to-1 multiplicity that includes two statements (the first two lines), and another with a 0-to-1 multiplicity that includes one statement (the last line).

Each statement includes one or more elements. An element can either be: (1) a fixed element, specifying the text that does not change within requirements, or (2) a variable element, serving as a placeholder that is filled to create requirements. Elements are defined by their name, their order within a statement, and whether they are optional. A fixed element is defined by its text. A variable element can be of three types: (1) *String*, i,e., a placeholder that can be filled with any value, (2) *Enumeration*, i.e., a placeholder that can be filled with one of the literals of an enumeration, or (3) *Range*, i.e., a placeholder that can be filled with a value within a range. For instance, the simple type definition template in Figure 3 includes one *String* variable element, i.e., `type name`, and one *Enumeration* variable element, i.e., `type` with four literals : `Numeric`, `alphanumeric`, `Alphabetic`, and `Enumeration`.

To ease the definition of templates, we provide a set of predefined templates that cover the specification of common parts of requirements template, namely conditions, actions, enumeration and ranges. A requirement template can include zero or more predefined templates. The predefined templates have the same structure as requirements templates. i.e., they includes blocks, statement and elements. These templates are defined in [4]. The simple type definition template in Figure 3 includes a predefined template for `value enumeration`, which defines the representation of enumeration literals.

2) User Interface Metamodel: We developed the UI metamodel presented in Figure 5 by analyzing the documentation of some common UI libraries such as JavaFX[1] and SWING[2]. As the figure shows, a UI component can either be: (1) a basic component, which represents elements such as buttons, labels and text fields, or (2) a container, which holds and organizes basic components. UI components are defined by their name, their position within the UI (i.e., their coordinates x and y), their size (i.e., their width and height), their visibility, and their text. It is important to note that the UI metamodel covers only a subset of existing UI components, as we are exclusively focused on the components necessary for our specification UI.

In the UI metamodel, a container can either be a shell or a group. The shell serves as the primary layer of the UI, holding both groups and basic components. A group forms a section within the shell, organizing one or more basic components. There are two types of groups: (1) new groups, which are created from scratch, and (2) predefined template groups, which are fully built and tailored to support the predefined templates described in Section III-B1. The latter are stored in a library. They are defined to ease UI generation by providing the structure for recurring predefined templates.

Each container includes one or more basic components. A basic component can either be: (1) an abstract button, which could be a check box button or a regular button; (2) an input component that allows users to enter information, such as a text field, a dropdown list, or a spinner; or (3) a label that accompanies an input component to guide the user on the information to be entered within the input component.

[1]https://openjfx.io/javadoc/23/
[2]https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/

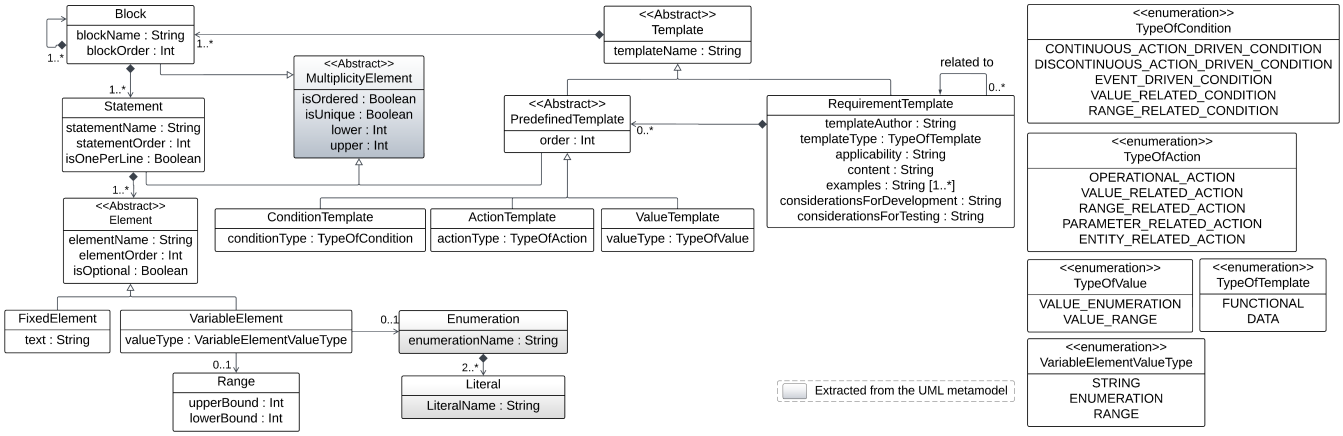


Fig. 4. The UTL metamodel [4]

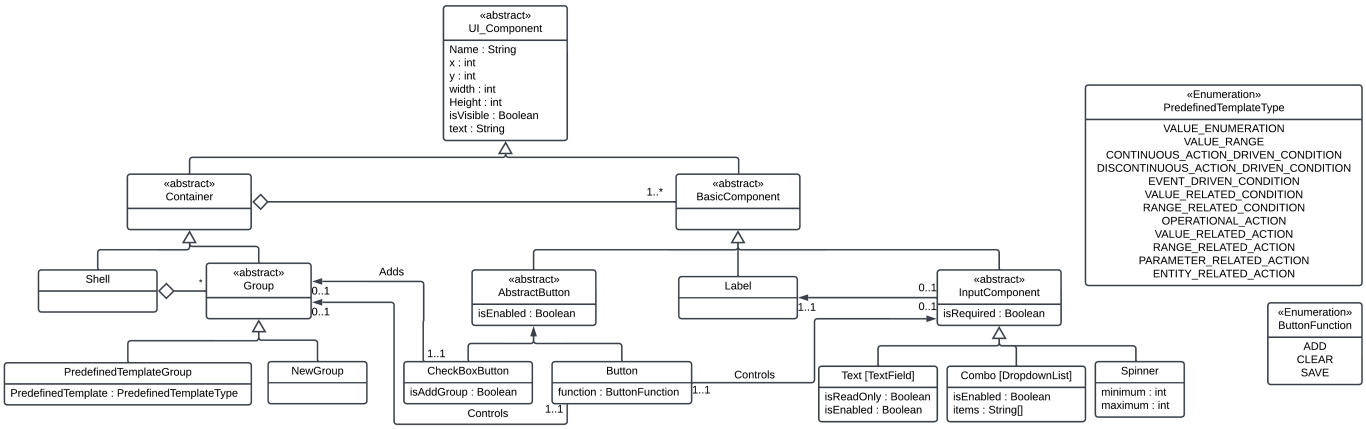


Fig. 5. The UI metamodel

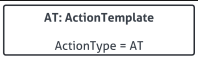
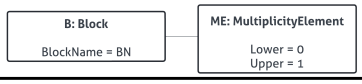
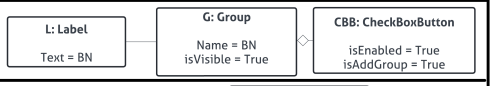
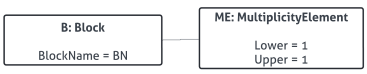
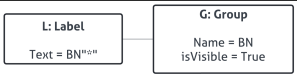
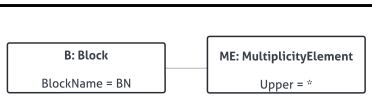
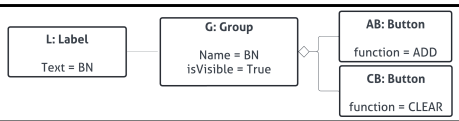
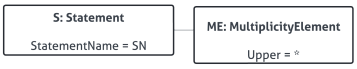
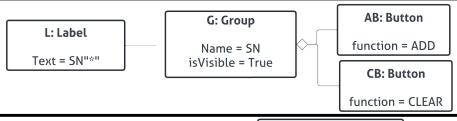
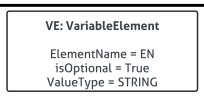
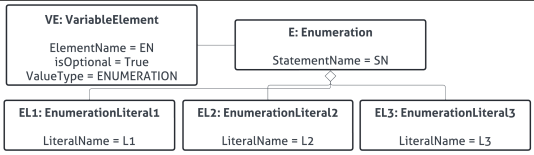
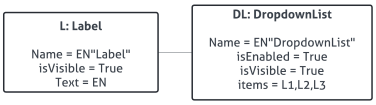
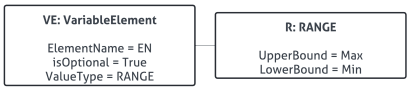
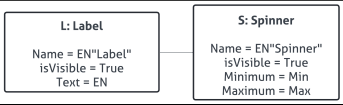
Each of these components is defined by specific attributes. For instance, a button can trigger a function for adding or removing certain groups from the UI. A dropdown list is defined by the items it contains, and the spinner is defined by the minimum and maximum values that constrain the input range.

3) Mapping Rules: We support the generation of UI from input templates through mapping rules between the template metamodel and the UI metamodel. Table I presents the mapping rules. UI components are identified differently depending on the type of the template element in question (i.e., predefined templates, blocks, statements, or variable elements). If the input template includes predefined templates, each is mapped to its corresponding UI group, which is predefined and stored in a library (R1, R2, and R3 in Table I). For instance, the `value enumeration` predefined template within the simple type definition template (Figure 3) is mapped to the `VALUE_ENUMERATION` UI group.

Each block is mapped to a group having the same name of the block, which is linked to a label that displays the block's name (R4, R5, and R6 in Table I). Depending on the multiplicity of the block, some UI components can be added to the group. If a block has a 1-to-1 multiplicity, no additional UI components are added to the group (R5 in Table I)). If a block has a multiplicity of 0-to-1, the group is linked to a checkbox that controls whether the block is visible in the UI (R4 in Table I)). Finally, if a block has an upper multiplicity of "many" (e.g., zero-to-many or one-to-many), the group will include two buttons: one that adds the entries of the group to the requirement's text, allowing multiple entries, and another that clears the block's text from the requirement's text (R6 in Table I). Consider the example of the simple type definition template in Figure 3. The first block (i.e., the first two statements) has a multiplicity of 1-to-1. Thus, the corresponding UI group will not include additional components. The second block has a zero-to-one multiplicity, and thus the group will be linked to a checkbox.

Once groups are defined for blocks, the next step is to identify the UI components for the contained statements. If a statement has an upper multiplicity of "many", a group will be created for it within the group defined for the block (R7 in Table I). Similar to blocks, the group will be linked to two buttons for adding and removing the group's entries to/from the requirement's text. Otherwise, no group is created, and the UI components are identified directly from the variable elements of the statement. For instance, the last statement of the composite type definition template (Figure 3) has a one-

TABLE I
MAPPING RULES

Component Type	Rule ID	Template metamodel	UI metamodel
Predefined Template	R1		
	R2		
	R3		
Block	R4		
	R5		
	R6		
Statement	R7		
Variable Element	R8		
	R9		
	R10		

to-many multiplicity. Thus, it is mapped to a UI group.

Finally, the last step is to identify the UI components for the variable elements included in the statements. It should be noted that the components are only identified for variable elements, as fixed elements remain the same for all requirements, not necessitating entries from users. Each variable element is mapped to a different input component depending on its type, but all input components are linked to a label displaying their name. For instance, if the variable element is a STRING, it is mapped to a text field, allowing the user to enter any value (R8 in Table I). If the variable element is an enumeration, it is mapped to a dropdown list that includes the literals of the enumeration (R9 in Table I). Finally, if the variable element is a range, it is mapped to a spinner that is constrained by the range's upper and lower bounds (R10 in Table I). For example, in the composite type definition template (Figure 3), the `text name` and `parameter name` elements are mapped to text fields, while the `variability` element is mapped to a dropdown list with two items: `FIXED` and `VARIABLE`.

C. Rendering and integration

The third step of our approach is *Rendering*, where the identified UI components are integrated into the existing specification window. Figure 2 presents the rendered specification window for the composite type definition template. It shows the common portion and the generated UI components for the template, i.e., the input components for type name, parameter name, and variability, along with the add and clear buttons. Finally, the fourth step is *Integration*, where the UI is connected to the backend code for the corresponding template. In our previous work [3], [4], each template is represented by a template model which is implemented by generating the code that supports the creation of requirements. In this step, the generated UI is linked to the generated template code. By the end of this step, the output is a fully functional UI enabling requirements specification using the input template.

D. Tool support

To support the automated generation of UI for template-based requirements specification, we extended our tool MD-

RSuT (Model-Driven Requirements Specification using Templates) [3]. MD-RSuT is an editor that supports the specification and management of requirements using templates. MD-RSuT includes JFIO (Just Fill It Out), an editor for UTL. JFIO supports the creation of templates using UTL [4]. Both MD-RSuT and JFIO are developed using the Eclipse Modeling Framework (EMF) as it supports the creation and evolution of models [11]. In addition, JFIO is developed using: (1) Xtext, a framework for the development of DSLs [12], to create the grammar and editor for UTL, and (2) Xtend, a high-level programming language for code generation [12], to automate the generation of code for the created templates.

We implemented our approach using: (1) EMF to build the UI metamodel, (2) Xtext to create the grammar and the editor for UI models, and (3) Xtend to automate the generation of the UI code. When a template is created using JFIO, both the backend code (generated from the template model) and UI code (generated from the UI model) are automatically generated for the saved template. The fully functional specification UI is then integrated into MD-RSuT and it is provided as an option for specifying requirements. Figure 2 presents an example of a UI that was generated by our tool.

IV. Evaluation

We evaluate our approach with regard to two questions:

- **RQ.1.** Do generated UIs adhere to UI design principles?
- **RQ.2.** How does our approach compare to the manual development of UIs?

To answer these questions, one of the authors applied our approach to generate UIs for two well-known templates, namely EARS Templates [13], and Rupp's template [14]. The generated UIs were then analyzed by the same author, and the analysis was validated by the other two authors.

A. RQ.1: Do generated UIs adhere to UI design principles?

Designing effective user interfaces is a challenging and complex task that requires careful consideration. In [15], Constantine and Lockwood provide a set of six principles for designing effective user interfaces. These principles aim to enhance the usability and user experience. The proposed principles are as follows [15]:

- *Structure:* UI components should be organized efficiently to facilitate user navigation through the UI.
- *Simplicity:* the tasks commonly performed by users in the UI should be simplified as much as possible.
- *Visibility:* only the elements that are needed by the user should be visible.
- *Feedback:* users should be informed about the system's state and the outcomes of their actions (e.g., errors).
- *Tolerance:* UIs should be designed to be tolerant and flexible by accommodating various user interactions and handling unexpected behaviors.
- *Reuse:* reusable components should be used to maintain consistency in the appearance and behavior of UIs.

We analyzed the UIs generated by our tool against these principles. Regarding the *Structure* principle, our approach supports: (1) the organization of UI components in a manner that reflects the structure of the input template (e.g., by maintaining the same order of elements as in the template), (2) the arrangement of UI components within the specification form, ranging from generic elements to specific ones, and (3) the grouping of closely related components (e.g., elements of the same statement). Regarding the *Simplicity* principle, our approach promotes a minimalist design including only necessary UI components for specifying the variable elements of the template. Additionally, it enables the specification of requirements in a single step within one window to reduce the cognitive load on the user. Regarding the *Visibility* principle, our approach ensures visibility of all the options, i.e., templates, that can be used for the specification (the *"Template type"* dropdown list). Furthermore, it provides access to a view displaying information about the template via the `"?"` button.

Regarding the *Feedback* principle, our approach displays the requirement's text reflecting the information filled in by the user in real time, allowing them to validate their entries. Also, once the specification is complete, a message is displayed to indicate if the requirement has been saved, or an error is detected. For the *Tolerance* principle, our approach minimizes errors by using spinners and dropdown lists that restricts entries to valid values. It also allows the user to clear parts of the requirement's text if they enter an incorrect value. Finally, for the *Reuse* principle, our approach maintains consistency among the generated UIs by reusing the same base window for the specification and adopting the same UI components for the same type of entries. Finally, as our approach is model-driven, it promotes reuse across developed UIs. Overall, UIs generated through our approach adhere to UI design principles.

B. RQ.2: How does our approach compare to the manual development of UIs?

The goal of this question is to investigate the advantages that our approach provides compared to manual UI development. In previous work [3], we manually built seven UIs for a set of seven templates. An example of a manually developed UI is provided in Figure 2. The UIs were implemented by two developers. This manual development process will serve as our manual reference for comparison.

In the manual process, we estimated a seven hour development for each UI supporting a template. This is because we were familiar with the libraries used to implement the UI. We can anticipate that the time would be longer for novice developers and developers unfamiliar with the programming language and used UI libraries. While with our approach for UI generation, the UI is generated and integrated with the rest of the tool in few seconds. Additionally, we effectively defined the UI and UTL metamodels, which can be directly reused to specify a wide range of new templates. This is a tremendous advantage of our approach as it reduces significantly the development time and efforts.

Our approach also promotes maintainability. In our prior manual process, the templates kept evolving. This required updating the UI accordingly, thus adding more time and efforts to

the development task. Our approach reduces the maintenance time and efforts considerably. It requires minimal efforts to generate new UIs through the reuse of certain structures (e.g., predefined templates) and components. This eases the creation of new templates and the evolution of existing ones.

C. Threats to validity

Some threats might impact the internal and external validity of our evaluation. For internal validity, the evaluation of the generated UI against UI design principles, and the comparison to manual generation was performed by the authors. This might introduce bias. To mitigate this, we adopted well-established UI design principles for the evaluation. Also, we plan to perform user studies with end-users to get more insights on the quality of the generated UIs, and the usability of the tool. With regards to external validity, the scope of the evaluation is limited. We only compared the UIs that we manually developed for our templates to those automatically generated. To address this, we plan to compare the UI generated by our approach to other that were developed by other practitioners/tools. Additionally, we only applied our approach on templates that we created ourselves and two others. We plan to evaluate our approach on broader range of templates.

V. Conclusion and Future work

In this paper, we propose a model-driven approach for generating UIs to specify requirements based on templates. The generation is supported through mapping rules that link elements of the templates metamodel to elements of the UI metamodel. Our approach is model-driven, which facilitates: (1) the design of UI for all types of templates, (2) the reuse of components across different UIs, (3) the management, evolution, and extension of the generated UIs, and (4) the generation of UI for multiple platforms if needed. The generated UIs are consistent, reusable, and scalable.

We provide a systematic process for the generation of UI from input template, which includes: (1) *Preparation* where the input template is expressed using UTL, (2) *Components identification* where the UI components are identified through the mapping rules, (3) *Rendering* where the UI is build based on the identified components, and (4) *Integration* where the developed UI is connected to the backend code. We implemented our approach in our tool MD-RSuT, enabling the automated generation of UI for new templates. This supports the development of UIs with minimal time and efforts.

We evaluated our approach by comparing it to manual development, and assessing the quality of the generated UIs. The evaluation confirmed that the generated UIs adhere to UI design principles, namely structure, simplicity, visibility, feedback, tolerance, and reuse. It also showed that our approach provides multiple advantages over manual development. As future work, we plan to perform user studies to further evaluate the quality of the generated UIs. We also aim to improve the generated UI's customizability and styling to better align with the user's needs. Finally, to make the specification UIs more dynamic and informative, we aim to support dependencies between UI components, and to accommodate the verification and auto-filling of requirements against domain knowledge.

References

[1] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, 1990.

[2] "Ieee recommended practice for software requirements specifications," *IEEE Std 830-1998*, 1998.

[3] I. Darif, C. Politowski, G. El Boussaidi, I. Benzarti, and S. Kpodjedo, "A model-driven and template-based approach for requirements specification," in *ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2023.

[4] I. Darif, G. E. Boussaidi, and S. Kpodjedo, "UTL: A Unified Language for Requirements Templates," *the 40th ACM/SIGAPP Symposium On Applied Computing, the Requirements engineering track*, 2025.

[5] K. Kolthoff, C. Bartelt, and S. P. Ponzetto, "Automated retrieval of graphical user interface prototypes from natural language requirements," in *Natural Language Processing and Information Systems Proceedings*, 2021.

[6] K. Kolthoff, "Automatic generation of graphical user interface prototypes from unrestricted natural language requirements," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[7] R. Juárez-Ramírez, C. Huertas, and S. Inzunza, "Automated generation of user-interface prototypes based on controlled natural language description," in *IEEE 38th International Computer Software and Applications Conference Workshops*, 2014.

[8] M. Elkoutbi, I. Khriss, and R. Keller, "Generating user interface prototypes from scenarios," in *IEEE International Symposium on Requirements Engineering*, 1999.

[9] A. M. Rosado da Cruz and J. Faria, "A metamodel-based approach for automatic user interface generation," in *Model Driven Engineering Languages and Systems*, 2010.

[10] A. R. Puerta, H. Eriksson, J. H. Gennari, and M. A. Musen, "Model-based automated generation of user interfaces," in *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, 1994.

[11] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. 2009.

[12] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd. 2016.

[13] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy Approach to Requirements Syntax (EARS)," in *2009 17th IEEE International Requirements Engineering Conference*, 2009.

[14] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam : Foundation Level, IREB Compliant*. Rocky Nook, 2011.

[15] L. L. Constantine and L. A. D. Lockwood, *Software for use: a practical guide to the models and methods of usage-centered design*. 1999.