Journal on Software and Systems Modeling - https://doi.org/10.1007/s10270-026-01406-5

UML4UI: UML as a Foundation for Model-Based and Model-Driven User Interface Engineering

Jean Vanderdonckt · Emilio Insfran · Abel Gómez · Silvia Abrahão

Received: date / Accepted: date

Abstract Model-Based and Model-Driven Engineering provide systematic means to manage the complexity of modern User Interface (UI) development by elevating models to first-class artifacts. Both approaches express multiple abstraction levels of the UI, while the latter uses model transformations to move between them. Both ensure consistency, improve reusability, reduce coding effort, and strengthen traceability across levels. However, several challenges remain. This paper examines the role of the Unified Modeling Language (UML) as a foundational modeling framework for both approaches. We show how UML can be used to (meta-)model presentational, behavioral, and interaction aspects across various abstraction levels, from task and domain models, through abstract and concrete UI specifications, and down to final UIs. *Forward engineering* produces UIs from models, *reverse engineering* recovers models from existing UIs, and *lateral engineering* adapts them to the context of use. Using a running example, we compare Model-Based and Model-Driven Engineering and derive some insights into the practical use of UML to address recurring challenges in UI development. We further provide a comparative discussion of UML-based approaches by reviewing existing work, namely UI Description Languages (UIDLs) as domain-specific modeling notations for UI development. We identify open research challenges that must be addressed to fully leverage UML as a comprehensive and standardized modeling and transformation framework for UI development.

Keywords Model-driven engineering · Model-based user interface engineering · Unified Modeling Language (UML) · User interface description language · User interface modeling.

Université catholique de Louvain, BE · Universitat Politècnica de València, ES

1 Introduction

When a designer, a developer, or any stakeholder [44] is asked to name some models to characterize software data, the answers come quickly and consistently: an object-oriented model represented using a class diagram of the Unified Modeling Language (UML) [54] or an Entity-Relationship-Attribute (ERA) model. If the same stakeholders are asked to identify a model that describes a process within a software system, they typically point to a UML activity diagram or to a Business Process Model and Notation (BPMN) diagram. However, when the question shifts to how user requirements can be modeled, they might mention a UML use case diagram, but usually offer little more than that. In this sense, the closer we move to the end user, the less likely it becomes that practitioners can name a well-established modeling notation or practice. Indeed, when asked to name a model characterizing the User Interface (UI) of a software system, most are unable to mention a single one. Some may mention UI toolkits, Integrated Development Environments (IDEs), or programming languages, but none of them truly correspond to an established model-based approach to UI development or engineering.

This situation contrasts with the maturity of modeling practices for data, processes, and architectures. For these aspects, standardized notations [60], metamodels and models [4], and development environments [25] have been widely adopted [74]. In comparison, the UI, despite being the primary interaction point between users and software, has often remained outside this systematic modeling discipline, mainly because of its complexity [73] that needs to be adapted to different users, platforms, and environments. In many development settings, it continues to be hand-coded or manually crafted,

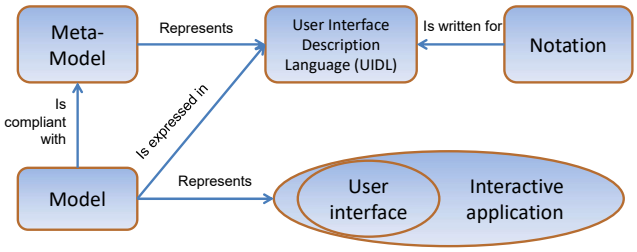


Fig. 1: A structured approach for UI development

even when the rest of the system is engineered through model-based or model-driven approaches.

Nevertheless, several initiatives, including both early [70] and recent [42], have introduced structured modeling approaches for UI development (Fig. 1). *Model-Based User Interface Development* (MBUID) [38] and *Model-Driven Engineering of User Interfaces* (MDE-UI) [68], itself sparked by Model-Driven Development [59], introduced a structured approach for developing UIs (see [51] for a comparison of approaches for UI generation), but in a different way.

Model-Driven Architecture (MDA) [47] is the OMG (Object Management Group) initiative that proposes to define a set of non-proprietary standard models that specify interoperable technologies with which to realize MDE with automated transformations. Both MBUID and MDE-UI involve the use of models, which entails that a modeling language must be compliant with a metamodel [19] to enable the metadata to be understood in a standard manner. This is a precondition for any activity to perform automated transformation. The main difference between MBUID and MDE is that the former does not necessarily entail transformation of models, while it is mandatory in the latter. These transformations are themselves regulated by a metamodel, which is absent in MBUID. The models representing the UI and its conceptual aspects [6] should be compliant with a metamodel represented with the help of a User Interface Description Language (UIDL), a domain specific language for UI using a standard notation such as XML [17] and preferably UML as a metamodel [60].

Although grounded in the use of models, MBUID and MDE-UI approaches have often relied on *ad hoc* or non-standard modeling languages and toolspecific representations. This situation contrasts with the principles of MDA, whose core premise is the definition of models at different abstraction levels and the use of standardized modeling languages capable of uniformly expressing system structure and behavior. Thus, UML plays a pivotal role in this context, as the primary OMG-endorsed modeling language, it provides a rich set of diagram types, a formally defined metamodel, and extensibility mechanisms that support domain-specific needs without sacrificing alignment with the standard. UML serves as the *lingua franca* for defining, relating, and transforming MDA models, ensuring consistency across viewpoints and enabling tool interoperability.

This paper investigates the role of the Unified Modeling Language (UML) [46] [58] as such a foundational approach to express models and metamodels for UI development in a standardized fashion, by introducing *UML4UI*. This is because UML is already extensively used to model data structures, processes, behavior, and interactions in software systems, and its extensibility mechanism—e.g., stereotypes—make it a strong candidate for supporting UI modeling in a systematic and traceable way. Our premise in UML4UI is that UML can provide a coherent modeling backbone across the levels defined by the *Cameleon Reference Framework* (CRF) [16], a framework for structured modeling of UI aspects that have been the subject of a W3C effort [27] for standardization. To this end, we make the following contributions:

1. *Conceptualization of UML across CRF levels.* We introduce UML4UI, which establishes UML as a foundational modeling framework for UI engineering across the CRF abstraction levels, from Task & Domain UI models to Abstract UI models, to Concrete UI models—which corresponds to CIM, PIM and PSM, respectively—and to Final UIs, which are the running UIs to be obtained.
2. *Engineering paths within the CRF.* In UML4UI, we define and illustrate the engineering paths that connect the CRF abstraction levels, including forward engineering (models to UI), reverse engineering (UI to models), and lateral engineering (i.e., adaptation across contexts of use or modalities). These paths operationalize UML4UI by making explicit the transformations and traceability relationships between models and the target UI.
3. *Comparative discussion of model-based and model-driven UI engineering.* From a UML perspective, we analyze the relationship between model-based and model-driven approaches, clarifying their respective roles, overlaps, and trade-offs along several dimensions, such as abstraction, traceability, transformation, and reuse.
4. *Illustration through a Virtual Polling System.* We provide a running example modeled across all CRF levels in forward engineering and realized in multiple UI variants. This use case illustrates the feasibility of the UML4UI approach, exemplifies possible engineering paths, and ground our comparative discussion in concrete modeling artifacts.

By aligning UML and related modeling approaches with the CRF, and grounding the discussion in a running example, this paper aims to bridge the gap between mainstream software modeling practices and UI engineering. In doing so, it seeks to contribute to a more integrated, systematic, and traceable approach to the development and evolution of UIs within MBUID and specially MDE-UIs.

The remainder of this paper is organized as follows. Section 2 introduces the CRF and its alignment with MDE concepts. Section 3 explains how UML can support modeling across the different CRF levels. Section 4 summarizes the most frequent engineering paths enabled by this approach. Section 5 illustrates the proposal through a *Virtual Polling System* case study. Section 6 provides a comparative analysis of model-based and model-driven UI engineering, followed by a discussion of related work in Section 7. Finally, Section 8 concludes with final remarks and open research challenges.

2 Background: Cameleon Reference Framework and its alignment with MDE

This section provides the conceptual foundations of our proposal by revisiting the CRF and examining its relationship with model-based, and especially, model-driven UI engineering from the point of view of UML and MDA. Together, all these approaches establish the abstractions upon which UML*4*UI builds.

2.1 Cameleon Reference Framework (CRF)

The Cameleon Reference Framework (CRF) [16] structures the UI development life cycle in terms of four levels of abstraction with their corresponding (meta-) models—e.g., Task & Domain, Abstract UI, Concrete UI, and Final UI—and the relationships between these levels. The CRF served as a common vocabulary within the UI engineering community to discuss and express different perspectives on a UI in a standardized way, which has been recommended in a W3C definition [27].

Fig. 2 shows how a simple search engine UI—i.e., the Google search page—composed of three widgets, can be structured according to these four levels: an *edit field* for entering the search string, a *push button* labeled "Google Search" to perform a standard search on the search engine, and a second *push button* labeled "I'm Feeling Lucky" for immediately open the first URL returned by the search engine. These widgets, located at the level of the *Final UI*, are controls of a graphical UI that is further expressed at the level of the *Concrete UI* in terms of a window containing

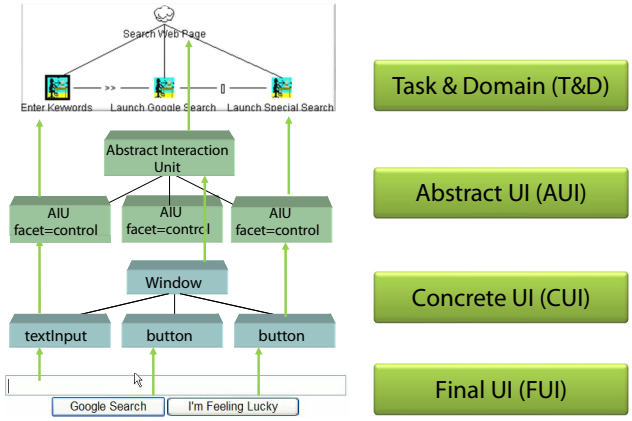


Fig. 2: The four levels of abstraction in the CRF

a text input and two buttons, which are named independently of any target platform. Independently of any interaction modality, these elements are considered as an *Abstract Interaction Unit* composed of three units with a facet of control, which are defined at the level of *Abstract UI*. Finally, and independently of any technological environment and adopting the viewpoint of the end user, these elements realize the task "Search Web Page" by decomposing the task into three interactive sub-tasks "Enter Keywords", "Launch Google Search", and "Launch Special Search". These latter elements are located at the level of *Task & Domain*, and are represented using the ConcurTaskTrees (CTT) notation [72].

2.2 CRF-compliant UI Modeling Approaches

UI modeling approaches have evolved through multiple research streams, particularly within MBUID [38] and MDE-UI [68]. Early MBUID approaches introduced explicit representations of tasks, domain concepts, abstract interaction objects [70], and concrete widgets, later structured according to the CRF. These approaches aim to separate concerns across abstraction levels and to support systematic derivation from high-level task descriptions to final UIs that are finally executable [18].

In parallel, MDE-UI has emphasized the use of transformation chains [8] to move between abstraction levels, typically relying on dedicated UI Description Languages (UIDLs). Languages such as UsiXML [37], UIML [33], MARIA [49], XIML [24], and IFML [13]—which has been the subject of an OMG recommendation [45]—provide domain-specific constructs tailored to UI concerns and often include automated transformations toward platform-specific implementations. Some UIDLs are particularly tailored for specific UI aspects, such as UIs of web services with WSXL [17], multimodal UIs with SMUIML [23], multi-platform UIs with PlastiXML

[20], and omni-channel UIs with OpenUIDL [42] (see Erazo et al.'s [25] survey for this purpose).

Despite these advances, the landscape remains fragmented. Many approaches introduce specialized languages or software support environments that are not tightly integrated with mainstream software engineering modeling practices. As a result, UI models are often treated as separate artifacts, loosely connected to domain, behavioral, or architectural models, which in practice hinders model consistency, makes traceability across abstraction levels more difficult, and limits the systematic automation that model-based and model-driven approaches aim to provide.

Similarly to UIDLs, other works have explored the use of UML to support UI modeling. Early approaches extended UML with stereotypes and profiles to represent web pages, navigation structures, and presentation elements, particularly in web engineering contexts, e.g., Conallen's *Web Application Extension* [21] and UML-based web engineering approaches [35]. Other proposals employed UML behavioral diagrams, such as activity or state machine diagrams, to model interaction flows and dialog control in interactive systems [75].

UML models have also been used as a basis for deriving user interfaces in MDA-oriented approaches, such as OO-H [32], which extends UML with specific views for modeling Web interfaces where UI elements are enriched with tagged values that support code generation, and the aforementioned *Interaction Flow Modeling Language* (IFML) [45], proposed by the OMG as a standardized notation for modeling user interactions that is closely aligned with UML and commonly used together with UML domain models as a basis for model-driven generation of user interfaces.

While these contributions demonstrate that UML can be applied to model specific UI concerns, they typically address particular domains, abstraction levels, or interaction aspects, rather than positioning UML as a unified modeling backbone across all CRF levels and engineering paths [37].

A similar situation can be observed beyond research-oriented UIDLs, where several industrial frameworks and platforms support cross-platform user interface development through higher-level abstractions. Examples include Flutter, React Native, and low-code platforms such as Mendix, OutSystems, and Microsoft Power Apps. These technologies enable developers to define user interfaces, component hierarchies, visual models, or domain-specific abstractions that can be deployed across web, desktop, and mobile platforms. Although these frameworks share key goals with Model-Based and Model-Driven UI Engineering, including reuse, productivity, and platform independence, they generally do not employ standardized modeling notations such as UML or explicit abstraction frameworks such as the CRF. Instead, they rely on proprietary component models, framework-specific DSLs, or platform-dependent runtimes. Consequently, traceability, interoperability, and integration with broader model-driven engineering processes remain limited, motivating continued research into standardized modeling and transformation approaches for the UI engineering lifecycle.

3 UML and MDA as a Foundation for CRF-level Modeling

Figs. 2 and 3 illustrate the four levels of abstraction in the CRF by emphasizing abstraction patterns [61] that align with the MDA terminology:

- The *Task and Domain* (T&D) level defines a model specifying the hierarchies of tasks that need to be performed on/with domain objects—or domain concepts—in a specific temporal logical order for achieving users' goals during the interaction with the UI. This level corresponds to the *Computation-Independent Model* (CIM).
- The *Abstract UI* (AUI) level defines a model that expresses the UI in terms of *Abstract Interaction Units* (AIU) [69], as well as the relationships among them. These AIUs are independent of any implementation technology or modality, e.g., graphical, vocal, tactile, haptic, gestural. They can be grouped logically to map logically connected tasks or domain objects. This level corresponds to a *Platform-Independent Model* (PIM).
- The *Concrete UI* (CUI) defines a model that expresses the UI in terms of *Concrete Interaction Units* (CIU). These CIUs are modality-dependent, but implementation technology independent, thus platform specific—i.e., MDA's *Platform Specific Model*, PSM. The CUI concretely defines how the UI is perceived and manipulated by end users.
- The *Final UI* (FUI) level defines a model that expresses the UI in terms of implementation technology dependent source code. A FUI can be represented in any UI programming language (e.g., Java, C++), mark-up language (e.g., HTML, JavaScript), or script language (e.g., Python). A FUI can then be compiled at design-time [53] or interpreted at runtime [10], particularly for run-time adaptation [76].

On the other hand, the relationships between the CRF models are as follows (Fig. 3):

- *Concretization*, or *Reification*, is an operation that transforms a particular model into another one of a

lower level of abstraction, until the final UI code is reached. CRF shows a four-step concretization process: the T&D level (task model and/or the domain model) is *concretized* into an AUI model, which in turn leads to a CUI. A CUI is then turned into a FUI, typically by means of code generation or interpretation techniques. Therefore, the FUI level cannot be concretized. The concretizations are graphically depicted as top-down blue arrows in Fig. 3.

– *Abstraction* is an operation that transforms a UI representation from any level to a higher level of abstraction. A FUI can be abstracted into a CUI mode, which is in turn abstracted into an AUI model to end up with a Task model. The abstractions are graphically depicted as bottom-up green arrows in Fig. 3.

– *Reflection* is an operation that transforms a model into another model at the same level of abstraction and for the same context of use. Unlike abstraction and concretization, reflection does not move vertically across CRF levels; instead, it represents an intra-level evolution of a model, such as a redesign, restructuring, optimization, or refinement. For example, at the Concrete UI level, a dialog-based interface may be reorganized into a tabbed layout while targeting the same users, platform, and environment. Similarly, at the Task & Domain level, tasks may be reorganized or their temporal relationships revised without changing the context of use. Reflection can therefore occur at any of the four CRF levels. They are graphically depicted as purple loops at any level.

– *Information* is an operation that transfers elements from the user, the platform, and/or the environment models to inform any subsequent level of abstraction. For example, elements of the environment model can inform or guide the definition of an AUI model, elements of the platform model can inform or guide the definition of a CUI model. These relationships are graphically depicted as dashed, red arrows in Fig. 3.

– *Context-aware adaptation* is an operation that transforms a description intended for a particular context of use into a description at the same level of abstraction, but aimed at a different context of use (Fig. 4). The adaptation is qualified as *context-aware* as it is performed depending on the constraints imposed by a target context-(target in Fig. 4) with respect to a source context. These operations are graphically depicted as bidirectional horizontal orange arrows between two contexts of use in Fig. 4.

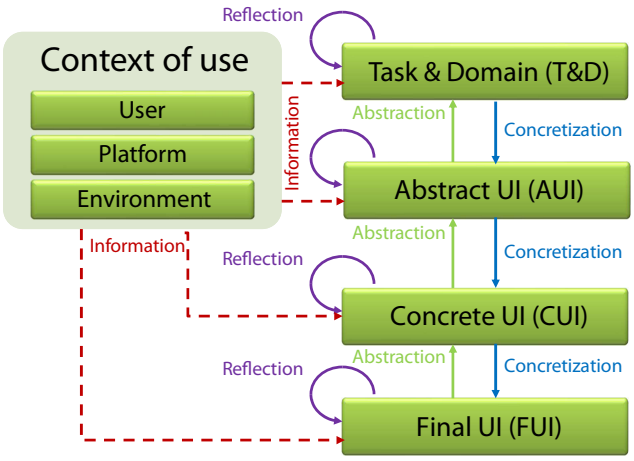


Fig. 3: The relationships between the four CRF levels

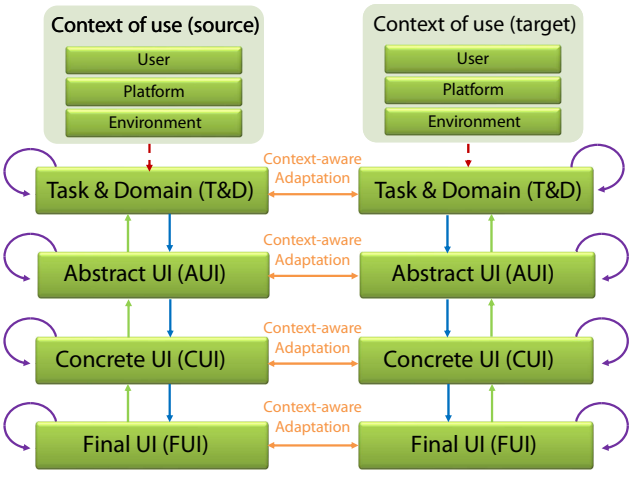


Fig. 4: The complete CRF framework

The aforementioned relationships always preserve some dimension, either *vertically*, i.e., concretization and abstraction, or *horizontally*, i.e., reflection and context-aware adaptation. To address non-vertical or non-horizontal transformations, *cross-cutting* [37] is a transformation of a model into another one at a different level of abstraction, higher or lower, while changing the context of use[1].

Similarly to the correspondence between MDA levels and CRF levels, a parallelism can be drawn between CRF models' relationships and MDA ones. As aforementioned, MDA explicitly structures software development as a set of model-to-model (M2M) transformations across abstraction layers, from CIM to PIM to PSM, which directly parallels the CRF's progression from T&D to AUI, CUI, and ultimately FUI. Concretization in the CRF mirrors the MDA notion of refinement toward platform-specific realizations, while abstraction corresponds to the inverse recovery of higher-

[1] Note that they are not represented in Fig. 4 for clarity.

level models from lower-level representations. Similarly, reflection within the same abstraction level echoes the MDA idea of intra-model evolution, and context-aware adaptation aligns with transformations between variants targeting different execution contexts or platforms. In this sense, the CRF's vertical and horizontal relationships can be naturally expressed using MDA's transformation semantics, thereby positioning UML and MDA as an appropriate foundation for capturing, formalizing, and transforming CRF-level models in a unified and interoperable way.

Finally, and orthogonal to the T&D, AUI, CUI and FUI levels, CRF makes explicit the context of use that may have an impact on the nature of the transformations used in the transformation process. The term *context of use* denotes an information space structured into three main models [16] (Fig. 3):

- The *User model* includes attributes and functions that describe the end user who is intended to use, or is actually using, the interactive system—e.g., a user profile, idiosyncrasies, a list of current tasks and activities. A particular attention is given to this model for a user with disabilities. A user model contains any attribute that may influence UI development, such as user preferences, user needs, level of experience, or frequency of use of platforms.
- The *Platform model* includes an integrated collection of software and/or hardware technologies and/or resource specifications that bind together the physical environment with the digital world. This model includes attributes and functions of the computing platform that may influence UI development, such as screen dimensions (if any), screen resolution, number of colours, vocal, gestural, haptic, and interactional capabilities.
- The *Environment model* includes spatio-temporal attributes, rules, and functions that characterize the physical and social places when/where the interaction will take place, or is actually taking place. This includes numeric and/or symbolic times and locations (e.g., in the morning, at 4 o'clock, at home, in a public space, on the move in the street, in the train or car), light and sound conditions, social rules and activities (e.g., hierarchical social organization, roles, spatial and temporal relationships).

Thus, a context of use is characterized by a triple *C=(U,P,E)*, where U, P, and E denote the *user*, *platform*, and *environment* models, respectively. For example, a business traveller rents a car while running in a busy airport on a smartphone. This stressful context is radically different from a context where the same user books a car on a desktop in a quiet office.

Although a context of use is mainly defined based on information about users, platforms, and environments, there are also other dimensions that can be relevant to characterize context and to properly adapt an software system. The application domain, for instance, can also add relevant information that complete the characteristics of the context of use. For example, in a safety critical environment, knowing that the interactive system supports air traffic control or a nuclear power plant provides useful information about the level of attention that is required from end users, depending on their level of experience. Fig. 5 shows an editing session of a context model in GrafiXML [40] where *U=any English-speaking person*, *P=web browser*, and *E=any stationary or mobile location*. These data will inform the rest of the UI development life cycle.

Considering the CRF dimensions, Fig. 6 shows a UML class diagram representing the top metamodels involved in UML*4*UI. All aforementioned models have their respective metamodels described as UML class diagrams, all inherited from a *model type* (*ModelType* in the figure). *ModelType* is the topmost superclass containing common features shared by all models, such as ID, name, creation date, modification dates, and corresponding CRF levels. To characterize a particular UI, a *UiModel* may consist of optional component models in any order and any number: not all these models should be incorporated. The *task model* (*TaskModel*) and *domain model* (*DomainModel*) represent the T&D CRF level. Similarly, the *abstract UI model* (*AuiModel*) and the *concrete UI model* (*CuiModel*) represent the AUI and CUI CRF levels respectively. Next, the *transformation model* (*TransformationModel*) and the *mapping model* (*MappingModel*) allow expressing the different relationships and the transformations among the four CRF levels; and the *context model* (*ContextModel*) allows describing the context of use. Finally, a *resource model* (*ResourceModel*) specifies the UI content, which may also be used to by transformation models to generate a FUI adapted to different contexts.

The *UiModel* is the core model of all child models, and, for example, the transformation model consists of a set of sequentially applied transformation rules. One transformation system applies one sub-derivation unit, defined as a collection of derivation rules that realize a basic development activity. We refer to appendices for the respective detailed metamodels: Appendix A.2 shows the metamodel for the task model structured according to UML constructs. Appendices A.3 and A.4 shows the metamodels for the domain model and the abstract UI model, respectively. Appendix A.5 shows the metamodel of a concrete UI for the graphical interaction modality or, in other words, for Graphical User

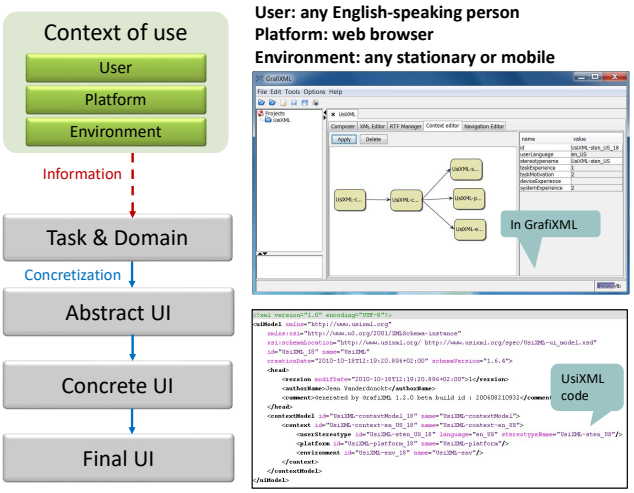


Fig. 5: Example of a context model

Interface (GUI). Any other interaction modality would require another metamodel, such as one for vocal interaction or gestural interaction. This metamodel could also be extended for addressing special requirements of safety-critical systems, complex systems [54], or real-time systems [64].

4 Engineering Paths

We define a *UI engineering path* [36] as a structured sequence of transformations that connects different abstraction CRF levels, such as from high-level models of user tasks and domain concepts down to a running final UI on a specific platform. The CRF can visualize any path, such as how a single set of user tasks and domain concepts can be transformed into multiple UIs [29] targeting different platforms [20], such as desktop, mobile, or voice-based systems. In essence, it represents the methodological route taken to move from abstract interaction requirements to an executable interface or the other way around.

For example, an engineering path may begin with a platform-independent task model, then branch into distinct AUIs tailored for graphical vs. vocal interaction, and finally evolve into CUIs and FUIs specific to technologies like web browsers or native mobile toolkits. This layered structure clearly illustrates how design decisions at higher abstraction levels influence, but do not rigidly determine, the resulting implementations.

Importantly, an engineering path is not limited to a single linear progression involving all levels. It may involve *forward engineering* (concretization from abstract to concrete models), *reverse engineering* (abstraction from implementation to higher-level models), *lateral engineering* (adaptation at the same level of abstraction), or *round-trip engineering* (hybrid and iterative movements between levels, such as back and forth paths). It can also branch, allowing one abstract specification to generate multiple CUIs or FUIs for different platforms. Finally, it can also skip a level, such as generating a UI directly from a task model [71]. Therefore, a UI development path is best understood as a formally describable chain of transformations across CRF levels [7] that defines how a multi-target UI is constructed or adapted.

Viewed from a broader model-based or model-driven perspective, engineering paths reflect the same disciplined, transformation-oriented mindset introduced earlier for UML and MDA. Just as MDA conceives development as a progression through explicitly related abstraction layers, the CRF organizes the evolution of models along structured levels and contexts that can be traversed in systematic ways. In both cases, the development process is understood as a model-guided refinement shaped by well-defined relationships between representations. This alignment allows engineering paths to be interpreted as straightforward model-based or model-driven processes depending on the levels of automation aimed or achieved, thereby offering a coherent

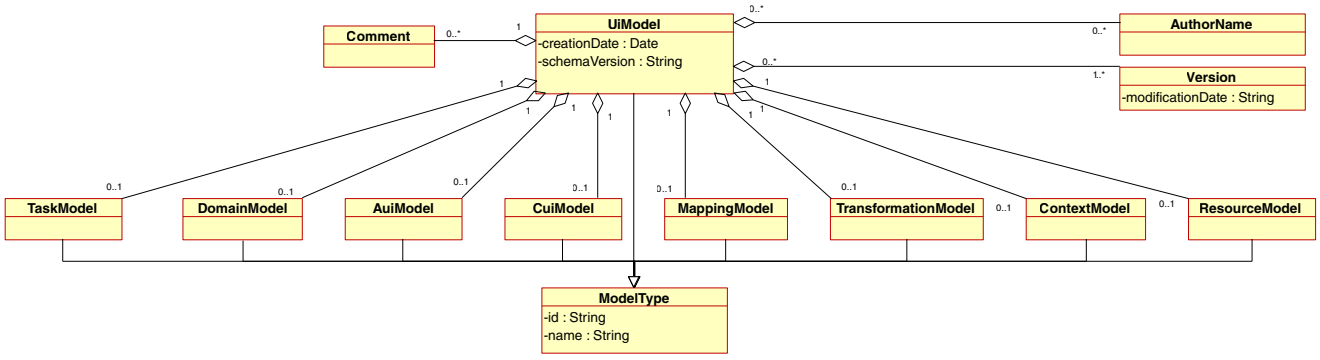


Fig. 6: UML Class Diagram for the top class *ModelType*, the core *UiModel* and its optional associated models

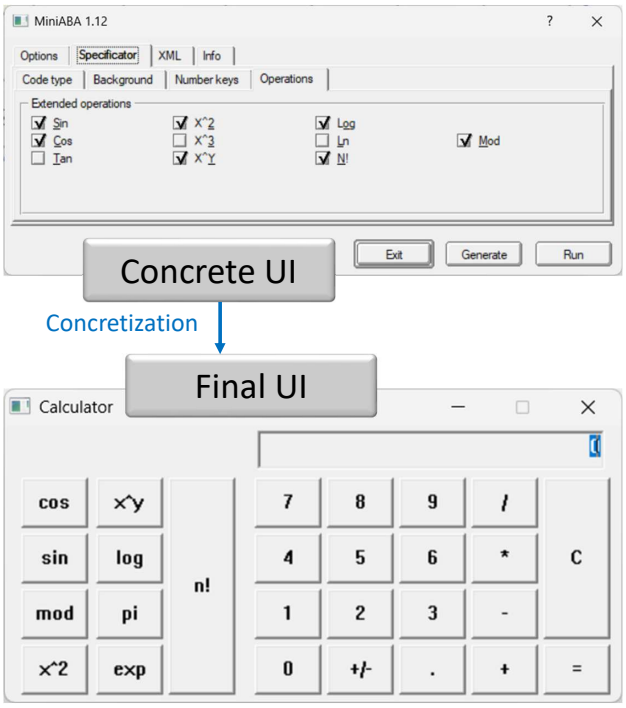


Fig. 7: Example of a forward engineering

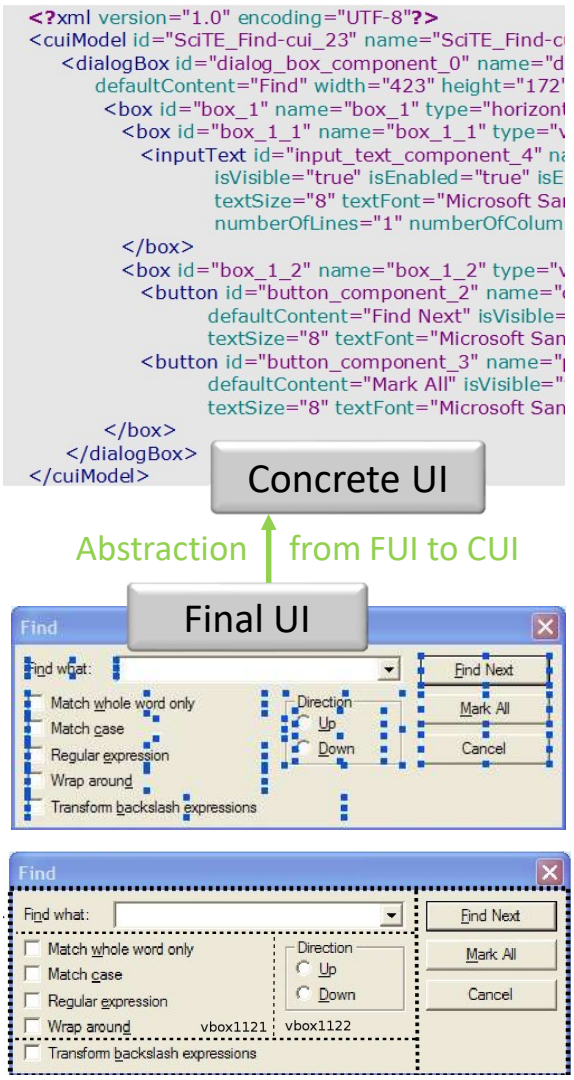


Fig. 8: Example of reverse engineering

conceptual link between the CRF abstraction structure and UML4UI.

The framework also accommodates hybrid and iterative paths (round -trip engineering): developers might move back and forth between AUI and CUI to refine usability, or partially regenerate only certain UI components while preserving others. By explicitly modeling these non-linear and branching trajectories, the CRF highlights that multi-target UI development [16,20] is not a single pipeline but a network of possible transformations, each with implications for UI quality factors, such as maintainability, portability, and consistency [34]. In what follows, we present some representative engineering paths that frequently arise in CRF-based development processes.

4.1 Forward Engineering Paths

In a strict forward path, practitioners begin with a high-level task and domain model, then derive an AUI that specifies interaction types without committing to widgets or layout. From there, platform-specific transformations generate one or more CUIs, which are finally rendered as FUIs using particular technologies, e.g., HTML/CSS, native mobile SDKs, or voice frameworks. For example, MiniAba [53] exploits a CUI model to automatically generate C++ code of a UI using feature modeling. Fig. 7 shows a CUI model of a calculator, from which three operations are removed: tan, X^3, and ln, that is then used to generate a final UI.

This staged refinement supports systematic multi-target development: a single abstract specification can branch into several CUIs, each optimized for device characteristics such as screen size, input modality, or interaction style. The CRF therefore illustrates how reuse and consistency can be maintained at higher levels while still allowing divergence at lower levels to accommodate platform constraints.

4.2 Reverse Engineering Paths

In reverse engineering scenarios, the CRF is equally valuable because it provides a structured and systematic way to “climb back up” from an existing implementation. For instance, the FUI of a legacy desktop appli-

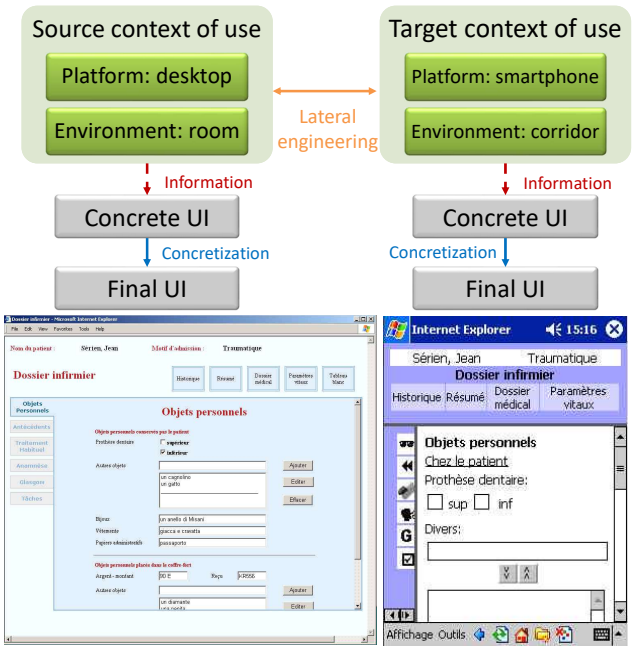


Fig. 9: Example of lateral engineering

cation can be analyzed to reconstruct its CUI structure and interaction patterns, which can then be abstracted into an AUI and task model. Once these higher-level representations are recovered, they can serve as the basis for retargeting the system to new environments [11], such as mobile [12] or multimodal platforms [65]. For example, UsiResourcer [50] extracts the resource files of a MS Windows GUI and creates a CUI model by abstraction (Fig. 8), to creating another GUI.

4.3 Lateral Engineering Paths

Lateral engineering refers to transformations that occur between artifacts at the same level of abstraction rather than moving up or down the CRF levels. For example, one CUI model designed for a desktop can be transformed into another CUI for a smartphone, e.g., by graceful degradation [26]. Fig. 9 shows how a source GUI for a medical doctor operating a desktop in the emergency room (left) can be transformed into a target GUI (right) for the same user operating a smartphone in a corridor. Similarly, adaptations may occur between different FUIs, such as migrating from one web framework to another while preserving the same interaction structure. Lateral engineering is particularly relevant for platform migration, technology updates, and device-specific optimization, as it enables reuse of existing design knowledge at a given abstraction level. Within the CRF, it highlights that multi-target development is not only about vertical refinement or abstraction, but also about horizontal adaptation across equivalent representations in different technological or contextual settings.

4.4 Context-aware Adaptation Paths

Context-aware UI adaptation refers to the dynamic or design-time modification of UI representations based on contextual factors such as user characteristics, device capabilities, and environmental conditions, that are captured in a context model. From the perspective of UML*4*UI, it can be viewed as a particular form of lateral engineering, where alternative UI representations at the same CRF abstraction level are selected according to the current context of use. When adaptation becomes context-aware, e.g., in collaborative MB-UID [31], the development path is no longer a reflection on the same level, but a horizontal relation to a target context of use (Fig.4). Fig. 10 shows how an initially designed GUI for route planning on a desktop (left) has been subject to context-aware adaptation to produce a GUI for a car browser (right). In line with recent work proposing conceptual reference frameworks for intelligent UI adaptation [2], in UML*4*UI we view lateral engineering paths as first-class citizens for systematically capturing context-driven UI variations across CRF levels. Context can directly influence transformations at multiple abstraction levels: at the AUI level, interaction modalities may be selected according to whether the user is operating in a hands-free situation or not; at the CUI level, layout and widget choices may adapt to screen size, e.g., using splitting rules [26], or input mechanisms; and at the FUI level, runtime adjustments may respond to changes in lighting, connectivity, or user preferences. By explicitly modeling context as an input to transformation processes, the framework enables systematic adaptation strategies rather than opportunistic UI adjustments. Consequently, the FUI can dynamically select and switch between alternative representations, which are stored in a *Resource Model*, based on information captured in the *Context Model*. This ensures that multi-target UIs [20] remain consistent across varying contexts of use.

5 A Running Example: Virtual Polling System

To illustrate UML*4*UI through a complete end-to-end example, we focus in this section on a forward engineering path that transforms the same T&D model into different CUIs and FUIs for multiple computing platforms and technological environments. We use a virtual polling system as a running example because forward engineering traverses all CRF abstraction levels

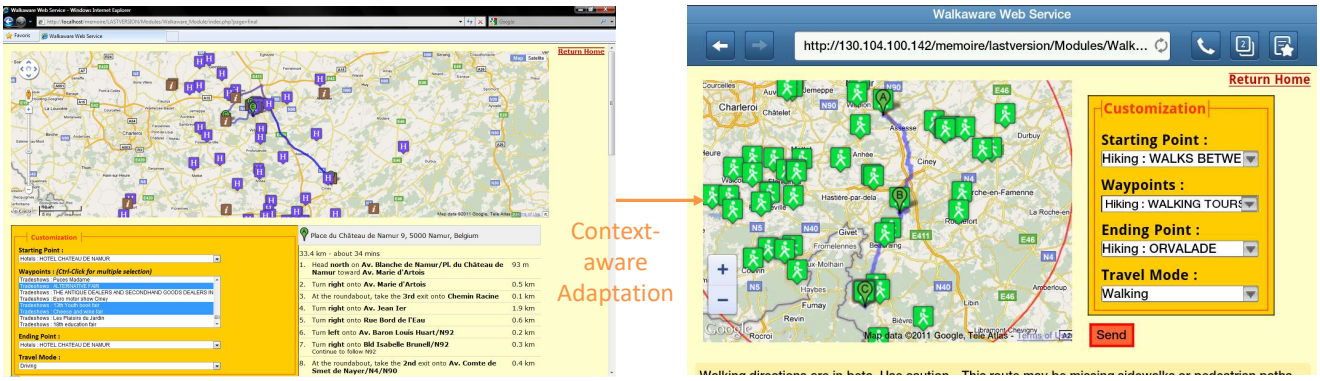


Fig. 10: Example of a context-aware adaptation: from a desktop (left) to a car browser (right)

and therefore provides the most comprehensive view of the modeling and transformation process. Reverse engineering, lateral engineering, and context-aware adaptation paths are illustrated separately in Section 4 through dedicated examples.

5.1 CIM Level: Task and Domain Modeling

Fig. 11 shows a task model using the ConcurTaskTree (CTT) notation, as used in MariaXML [49] and in the W3C MB-UID report [27]. To "Participate to Opinion Poll", the end user will have to "Provide Personal Data" and then enter a series of questions/answers, which is represented as a repeated sub-task "Answer Question", before finally "Send the Questionnaire". For each multiple choice question, the system "Shows a question" and the end user has to "Select answer(s)". Fig. 11 also shows the mappings between elements of the *task model* and elements of the corresponding *domain model*, also expressed as a UML class diagram. For example, the sub-task "Provide Personal Data" is directly mapped to the class "Participant". This mapping is relative: any modification to the "Participant" class will automatically be reflected in the lower abstraction levels, without affecting the *task model*. These mappings are recorded in a *mapping model* (Fig. 6) that is typically found in MDE-UI, but not in MB-UID. When the task model changes, the mappings to the domain model do not change at first glance, unless the *task model* is extended, which is represented in Fig. 12.

Instead of being directly forwarded to a predefined questionnaire, the end user can "Choose a Poll" by selecting the corresponding questionnaire, which is itself decomposed into questions and propositions of answers. IdealXML (Fig. 13) enables the designer to encode these models and the mappings of different types between, which is referred to as the mapping problem [43].

5.2 CIM to PIM Transformation: From Task and Domain to Abstract UI

Transforming a T&D model into an AUI involves mapping user goals and task structures to *abstract interaction units* and *objects* that are independent of any interaction modality. This model-to-model (M2M) transformation is performed by executing relevant transformation rules encoded in the *transformation model*. After this transformation has been executed, mappings between the source and target models are preserved in the emphmapping model for traceability.

The task model describes what users need to accomplish—e.g., selecting an item, entering data, confirming an action—and how tasks are structured—e.g., sequential, parallel, optional, iterative—thereby leading to the identification of abstract interaction units structure, followed by a selection of its components—see Fig. 14. During transformation, each interactive task is associated with an *abstract interaction unit* (AIU)—which is itself decomposed into sub-units, to end up with *abstract interaction objects* (AIO) [70]—such as input, output, selection, navigation, or command activation. The result is an AUI that specifies the logical organization of interaction units, dialogue flow, and grouping of abstract components, preserving the semantics and temporal relationships of the original task model while prepares the design for subsequent refinement into a CUI tailored to a specific platform associated to a particular interaction modality.

AIUs are graphically represented as containers—UML classes—encompassing parts of the task model to delineate their scope. This container simply means that these abstract elements, which are platform-independent, will be then mapped to concrete elements, which are platform-specific, according to the same structure when this transformation will be executed, i.e., the PIM to PSM transformation.

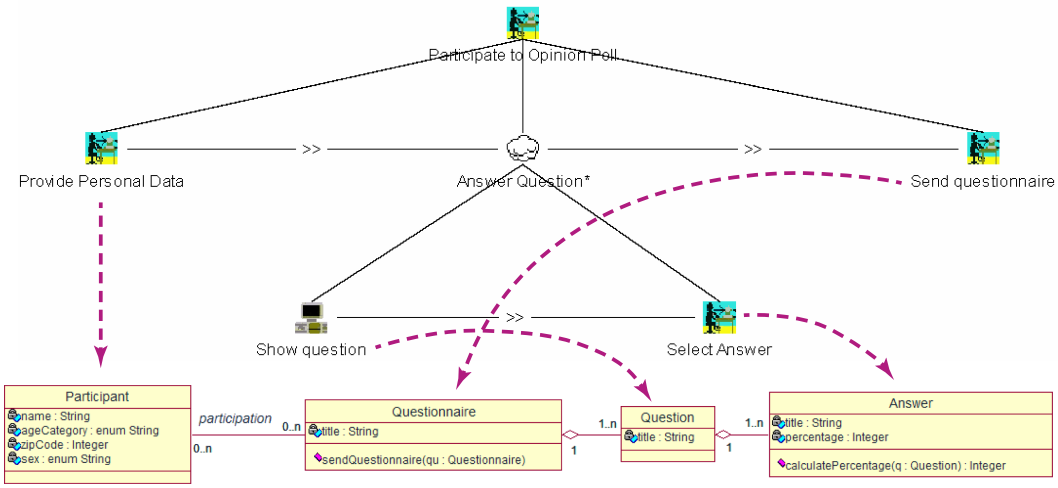


Fig. 11: Task model of the virtual polling system linked to the domain model

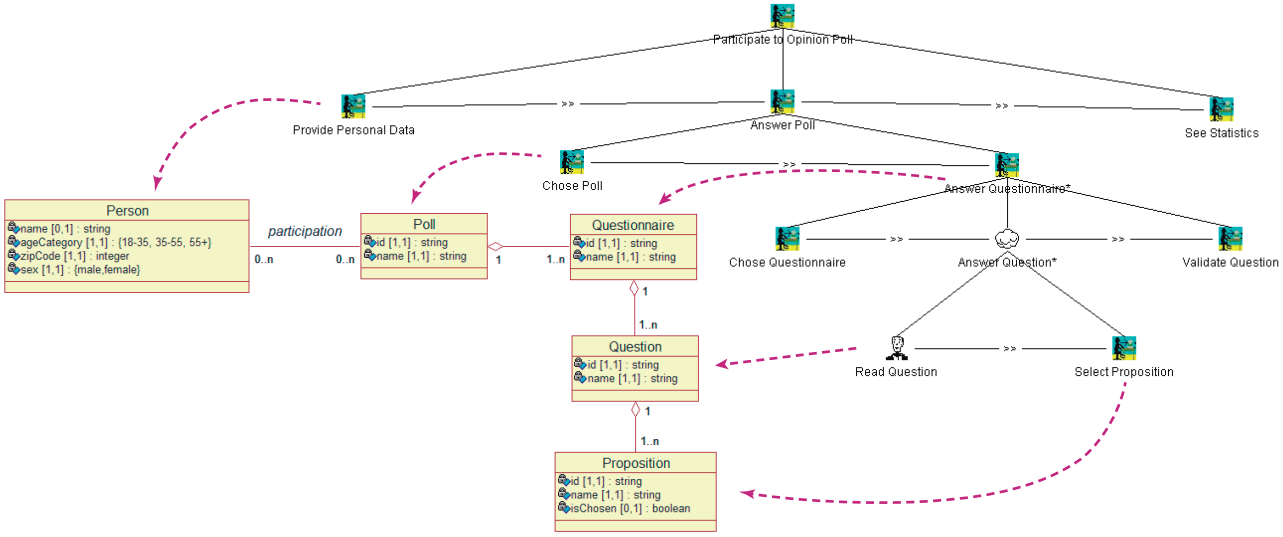


Fig. 12: Task model of the virtual polling system linked to the extended domain model

Fig. 15 shows an example of the identification of these elements based on the Task Model: a global container (AC1) covers all sub-tasks of the task model of our case study. Two main sub-containers, materializing two sub-units, are determined for the two first sub-tasks (AC11 and AC12). Abstract interaction objects (AIC) can be then identified according to high-level design options: maximal selection (i.e., the whole task model will be transformed into a single unit), minimal selection (i.e., each leaf of the task model will be transformed into an individual unit, which is appropriate for novice users who prefer wizard-like UIs), input/output selection (i.e., when the sub-tasks of a parent task are separated into two units: one for input and another one for output), or work-based selection (i.e., when the decomposition of a task into sub-tasks is guided by the work).

Once the AUI structure is identified—first and second steps in Fig. 14 on the following page—the subsequent steps are performed, such as the spatio-temporal arrangement of these units and their abstract interaction objects.

Fig. 16 shows two potential spatio-temporal arrangements using a Class Diagram for our running example, where a navigation across elements is specified, and relationships among these elements are created. The stereotypes *abstract adjacency*, *dialog control*, and *repetive*, help to specify this information. The first option (left) uses the nested UML notation for the composition relationship, where the elements of the unit corresponding to interaction with the questionnaire should

be rendered as closely as indicated in the diagram. The second option (right) focuses primarily on the structure of each container and the temporal aspect using the predefined stereotypes indicated above.

A MDE-UI environment [67] enables the designer to explore various design options to determine an AUI that will preserve desired user experience (UX) quality factors as recommended by [3]. This environment leverages the task hierarchy and dependencies defined in a task model to infer AIUs and their composition [14], taking into account task relationships such as sequencing, choice, and concurrency. By doing so, the method supports generation not just of a canonical AUI, but of all AUIs that are valid with respect to the task model semantics. These results can then be filtered or ranked according to metrics like cognitive load, efficiency, or platform constraints before refining them into CUIs and FUIs.

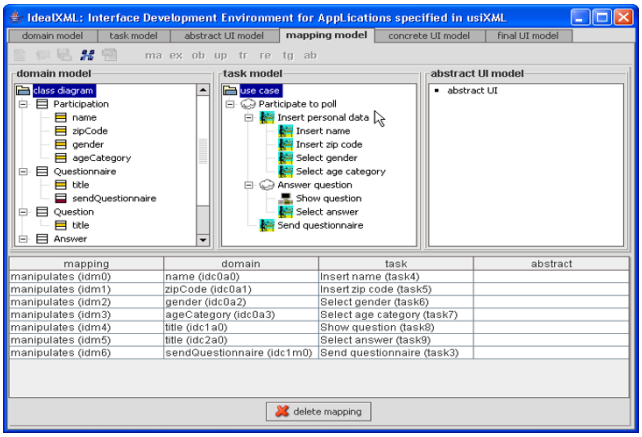


Fig. 13: Task model mapped to an AUI [43]

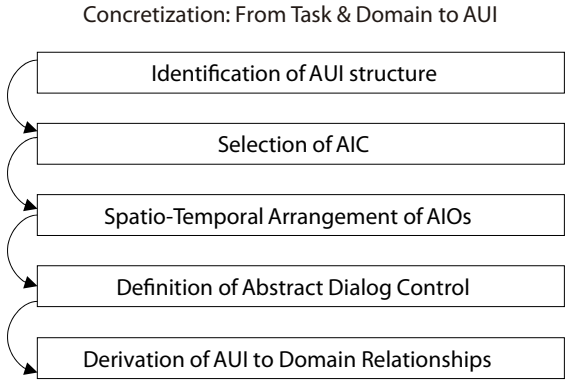


Fig. 14: Task model transformed into an AUI

5.3 PIM to PSM Transformation: From Abstract UI to Concrete UIs

Transforming an AUI into a set of CUIs involves mapping abstract interaction objects to platform and language-independent widgets and layout constructs. This M2M transformation can be performed by executing relevant transformation rules encoded in a *transformation model*, which result in mappings between the source and target models stored in the *mapping model* to preserve traceability. At the AUI level, elements such as abstract input, abstract selection, or abstract command define interaction intent without prescribing implementation details. During the transformation, these elements are mapped to concrete components that are abstracted for a particular modality, e.g., graphical, independently of the target platform. Appendix A.5 shows the metamodel for the graphical modality at the CUI level. If the interaction modality changes, say a gestural interaction, then the corresponding metamodel will change. Similarly to the process of creating an AUI, this transformation covers the following sub-steps—cf. Fig. 17: reification of abstract units into concrete ones (e.g., mapping an abstract unit to a window, a web page, or a dialog box), selection of concrete units (e.g., mapping a bounded numerical input to a spin button), arrangement of these units (e.g., creating a layout), defining a navigation (i.e., how to navigate), etc.

For example, Fig. 18 represents the concretization of an AUI intended to exploit a GUI on a desktop platform. In this case, the overall abstract interaction unit is mapped to a single window, that contains two concrete interaction units: one for the personal data and another one for the questionnaire. At this level, we decided to concretize the AUI into a CUI using a graphical modality of interaction (GUI) running on a desktop, but we do not know yet which computing platform will be targeted with which operating system, two decisions that are further delegated to the last transformation.

5.4 PSM to Code Transformation: From Concrete UI to Final UIs

Transforming a CUI into a FUI involves mapping concrete interaction objects to platform and language-specific widgets and layout constructs. This model-to-code (M2C) transformation can be performed in two different ways: by code generation and compilation, or by model interpretation.

In both cases, mappings are established between the CUI model and the target FUI elements stored in the *mapping model* to preserve traceability. The concrete

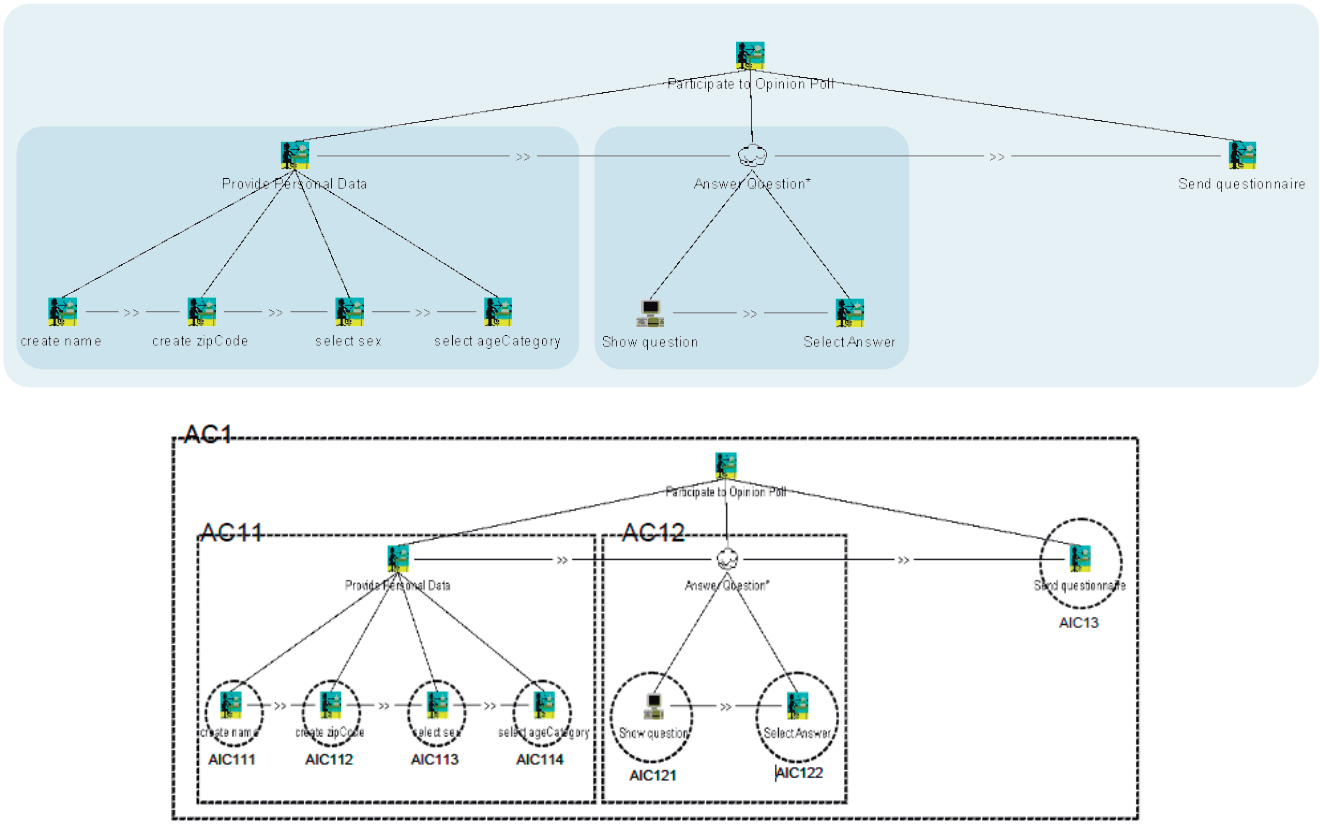


Fig. 15: Identification of the AUI structure

objects are thus mapped into elements elements available in the target programming environment.

For instance, an abstract text input may map to an HTML `<input type="text">`[2] element in a JavaScript-based web application, to a `JTextField`[3] in a Java Swing interface, or to a `TextBox`[4] in C# Windows Forms. Similarly, an abstract command action may map to an HTML `<button>`, a `JButton`[5] in Swing, or a `Button` control in Windows Presentation Foundation. Layout groupings defined abstractly in the AUI can be mapped to CSS layout containers (e.g., `Flexbox` or `Grid`), Java layout managers (e.g., `BorderLayout`, `GridLayout`), or XAML[6] stack panels. Through these systematic mappings, multiple graphical CUIs can be generated from the same AUI, each respecting the idioms, widget sets, and architectural constraints of its target programming language while preserving the interaction structure defined at the abstract level.

For example, Fig. 19 shows a screenshot of a Final UI programmed in Visual Basic[7] for MS Windows resulting from code generation based on the CUI sketched in Fig. 18: the overall concrete interaction unit is mapped to a window and the two sub-units are mapped to a group box contained in that window. Individual widgets have been selected based on data properties associated with attributes of the *domain model* [70].

Similarly, Fig. 20 shows how a CUI model is interpreted at run-time—as in [10]—in a rendering engine that produces real 3D UIs. In this rendering, a push button is really rendered with animation effects that give the feeling of really pressing a button. Similarly to the Visual Basic M2C transformation, the concrete interaction unit is mapped to a 3D window that can be flipped, rotated, etc., inside which all fields are concatenated. Questions and answers are rendered underneath in a sequence, a set of 5 questions/answers at a time.

[2] `https://www.w3schools.com/tags/att_input_type_text.asp`
[3] `https://docs.oracle.com/javase/8/docs/api/javax/swing/JTextField.html`
[4] `https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.textbox?view=windowsdesktop-10.0`
[5] `https://docs.oracle.com/javase/8/docs/api/javax/swing/JButton.html`
[6] `https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/`
[7] `https://learn.microsoft.com/en-us/dotnet/visual-basic`

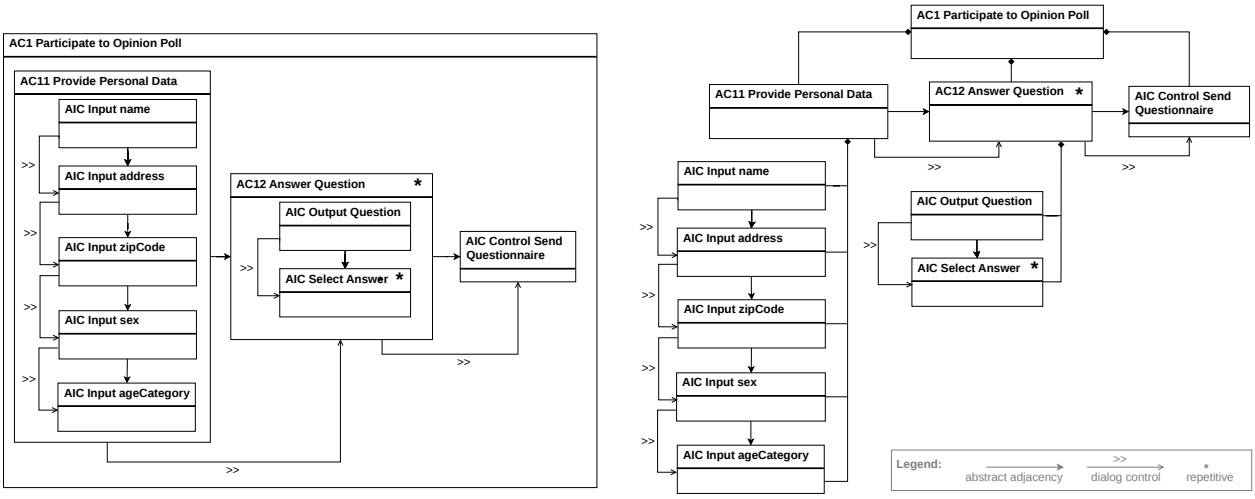


Fig. 16: Spatio-temporal arrangement of the abstract elements of the AUI

Finally, Fig. 21 shows that two concrete interaction units have been mapped to two subsequent screens implemented using the Virtual Reality Markup Language (VRML)[8] language, to produce a virtual reality UI ren-

[8] https://en.wikipedia.org/wiki/VRML

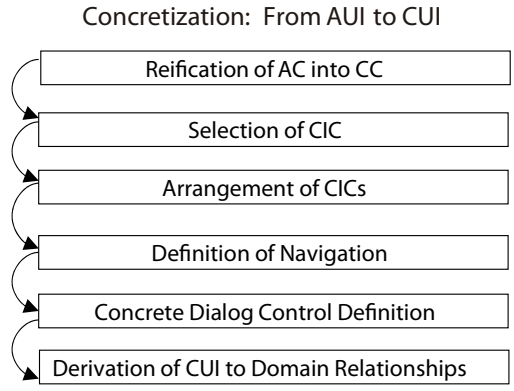


Fig. 17: AUI transformed into a CUI

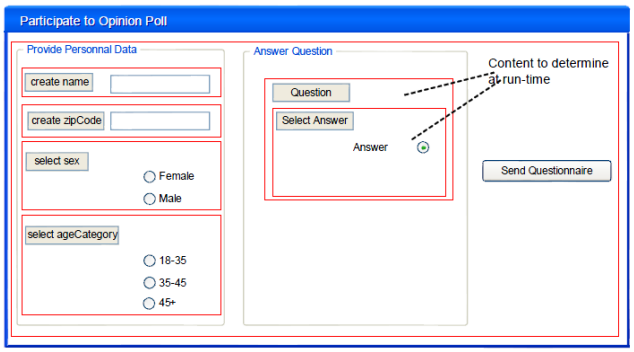


Fig. 18: CUI for a GUI on a desktop

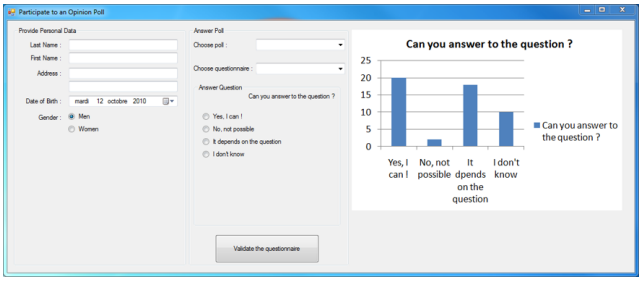


Fig. 19: FUI in Visual Basic for MS Windows

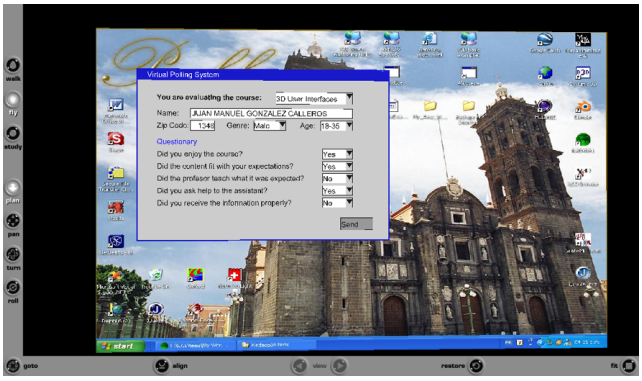


Fig. 20: Final UI in a 3D rendering engine

dered in a browser. The personal data are directly manipulated in the first screen—note that data have been mapped to avatars—while the question/answer part is mapped in the second screen. In this case, the FUI is split in two screens in virtual reality. This decomposition can be subject to other transformations: Fig. 22 shows a FUI implemented in X3D[9], the successor of

[9] https://en.wikipedia.org/wiki/X3D

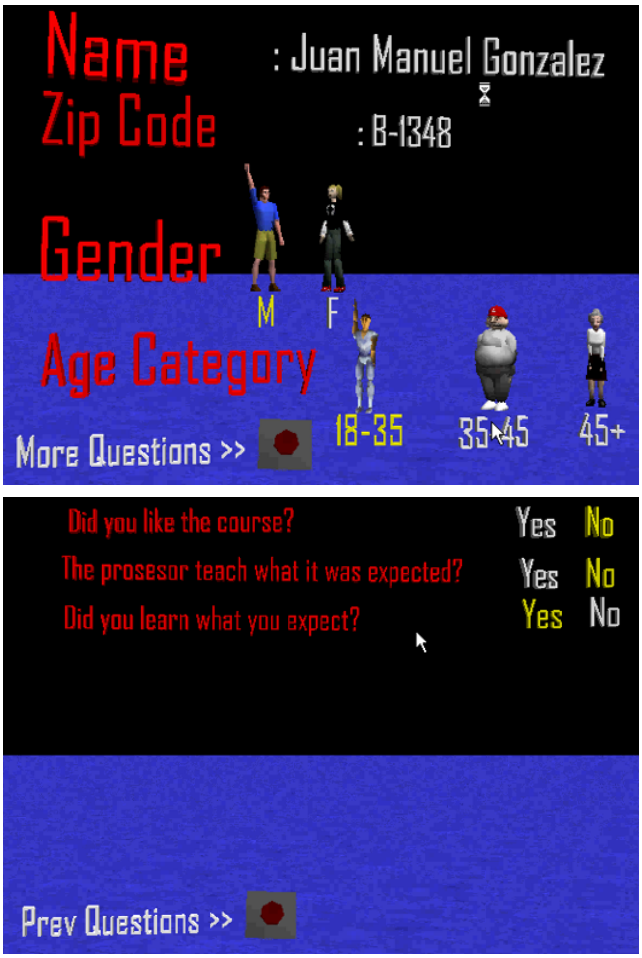


Fig. 21: Virtual Reality UI

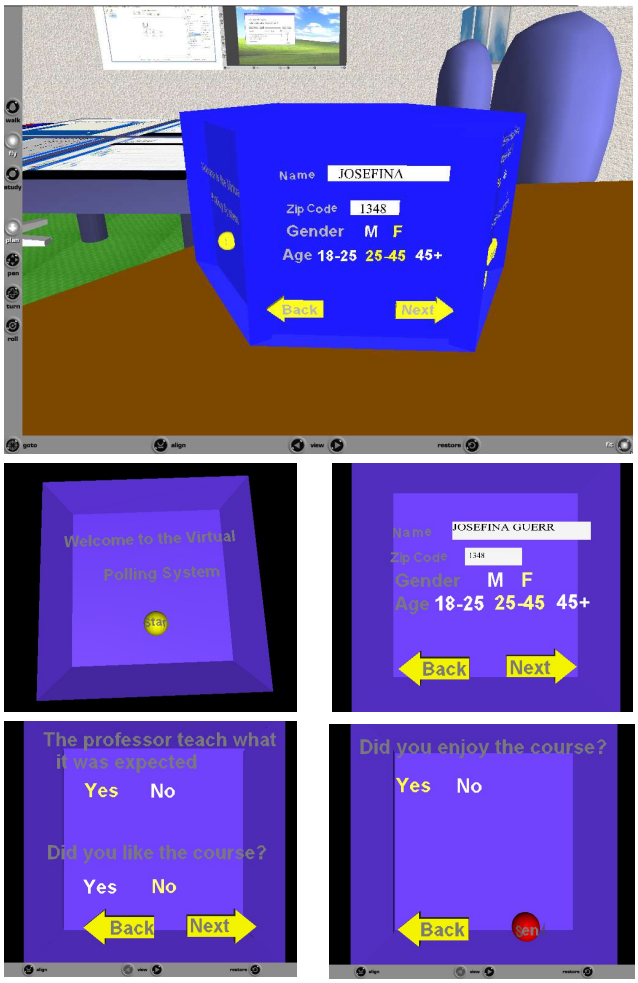


Fig. 22: Final UI in X3D

VRML. This FUI is also rendered in three dimensions and maps the CUI to a single scene, where each unit is mapped onto faces of a prism. The navigation between the faces of the prism is ensured through arrows indicating "Next" or "Previous" face. Figs. 19 to 22 are all representative examples of a GUI for the graphical modality, but developed in different toolkits and languages. Let us now consider other FUIs not exclusively involving the graphical modality.

Transforming a CUI into a FUI implemented using XHTML+Voice[10]—commonly X+V, an XML language for describing multimodal UI—involves mapping [66] abstract interaction objects to speech-based prompts, grammars, and multimodal synchronization structures, while allowing different vocal design variants. At the AUI level, elements such as abstract input, selection, confirmation, or command remain modality-independent. During transformation, an abstract selection task may be mapped to a *VoiceXML* `<prompt>` combined with a `<grammar>` defining allowable spoken choices, synchronized with graphical highlights in the `X` (visual) channel. An abstract input may become a vocal form field using `<field>` and speech recognition grammars, possibly paired with a visual text field for redundant or complementary input. Different variants can be generated depending on design strategy: a *voice-dominant variant*, where speech drives the dialogue and graphics provide feedback; a *graphical-dominant variant*, where voice acts as a shortcut or accessibility layer; or a *redundant multimodal variant*, where prompts are both spoken and displayed simultaneously. Each variant maps the same AUI interaction structure into different configurations of VoiceXML components, synchronization mechanisms (e.g., event handlers linking speech and GUI actions), and dialogue management strategies, while preserving the semantic intent and task flow defined at the abstract level. The main advantage of implementing the FUI in X+V is that the end user can choose which interaction modality is preferred: totally graphical (Fig. 23-left, where all widgets rendered

[10] `https://en.wikipedia.org/wiki/XHTML`

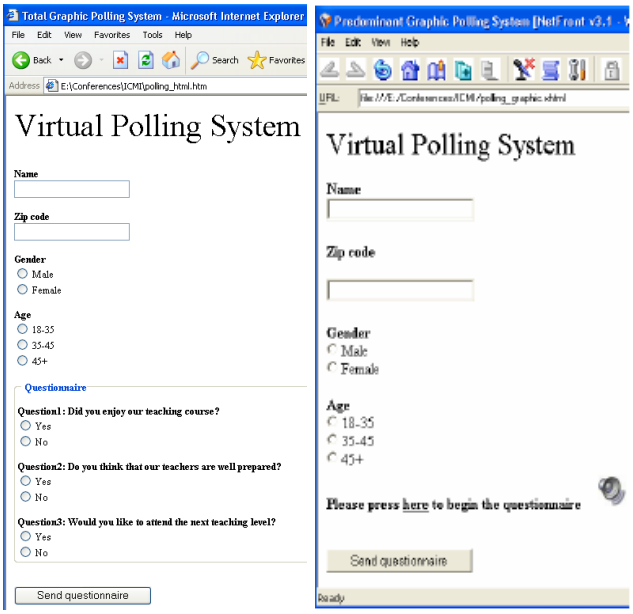
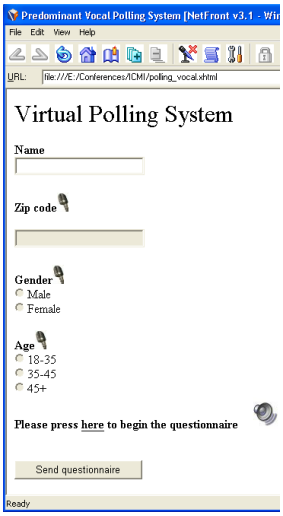


Fig. 23: FUI in X+V (part 1 of 2)

in a web browser can exclusively be manipulated by a pointer and a keyboard), predominantly graphical (Fig. 23-right, where the participant data are entered graphically and the answers are issued vocally), predominantly vocal (Fig. 24-left, where the end user can choose to enter data either graphically or vocally by speech-to-text transformation (note that only the last name cannot be entered vocally since this is language dependent), or totally vocal (Fig. 24-right, where a vocal session is initiated asking one question to the user at a time and waiting for the answer to be processed by speech-to-text).



C: Welcome to the virtual polling system.
C: Please say your name.
U: John Maverick.
C: Your answer was [John Maverick].
U: Yes.
C: What is your zip code?
U: 18.
C: Are you sure?
U: Yes.
C: Please say your gender.
U: Male.
C: Are you sure?
U: Yes.
C: How old are you ?
U: 32.
C: Are you sure?
U: Yes.
C: Please answer to the following questions
C: Did you enjoy our teaching course?
U: Yes.
C: Do you think that our teachers are well prepared?
U: Yes.
C: Would you like to attend the next teaching level?
U: No.
C: Do you want to send the questionnaire?
U: Yes
C: Are you sure?
U: Yes.

Fig. 24: FUI in X+V (part 2 of 2)

All aforementioned examples—from Fig. 11 to 24—represent different development paths in forward engineering that can be also be performed in UML4UI: starting from the T&D model—equivalent to the CIM level—transforming it into an AUI—equivalent to a CIM-to-PIM M2M transformation—and into a CUI—equivalent to a PIM-to-PSM M2M transformation—that is ultimately converted into different interaction modalities: graphical, vocal, or mixed.

6 Comparative Analysis: Model-Based vs. Model-Driven UI Engineering

The previous section introduced UML4UI and illustrated its application of forward engineering across CRF levels through a running example. We now turn to a comparative analysis of model-based UI development and model-driven UI development through this lens. Although both approaches elevate UI models to first-class citizens, they differ in their treatment of abstraction, transformation, and lifecycle integration. Grounded in core MDE principles, particularly those emphasized by Selic [55,56], this section evaluates these differences using a set of engineering-oriented criteria.

6.1 Comparison through selected criteria

To compare MB-UID and MDE-UI through the lens of UML4UI, we adopt evaluation criteria grounded in foundational principles of MDE. Selic argues, in his work on the pragmatics of model-driven development [56] and on the transition from model-driven development to model-driven engineering [57,60], that models must serve as primary engineering artifacts rather than mere documentation. According to this perspective, models should enable systematic abstraction, explicit transformation, automation, lifecycle integration, and rigorous reasoning.

This view has shaped the broader MDE research agenda, where abstraction and transformation are recognized as key mechanisms for managing complexity in large-scale systems [28]. More recent analyses of the state of the field have reaffirmed that issues such as language expressiveness, model management, transformation usability, traceability, tool robustness, and evolution remain central challenges for the MDE community [15]. These challenges closely align with the properties that determine whether models genuinely function as engineering artifacts. Selic's work [58,60,59] shows that quality models must exhibit quality factors such as abstraction, accuracy, predictiveness, understandability, and cost-effectiveness, and that MDE aims for au-

tomation and transformability from high-level models to running artifacts

In line with this engineering view of models, we evaluate the approaches using the following criteria: *(i)* expressiveness across abstraction levels, *(ii)* traceability, *(iii)* transformation effort, *(iv)* reusability, *(v)* tool support, *(vi)* change impact, and *(vii)* analyzability. These criteria reflect core properties that determine whether modeling approaches truly support complexity management, systematic development, and disciplined evolution across abstraction levels. Rather than assessing isolated language features, our comparison focuses on the extent to which each approach operationalizes these MDE principles within the specific context of user interface engineering.

6.1.1 Expressiveness Across Levels

Selic [62] argues that abstraction is the central mechanism by which models reduce accidental complexity [54, 57]. A modeling language must therefore be sufficiently expressive to represent the essential structural and behavioral properties of a system at different abstraction levels [61].

In MB-UID, expressiveness is often achieved using specialized notations tailored to tasks, abstract UI structures, or concrete UI elements. However, this expressiveness may be fragmented across heterogeneous formalisms. MB-UID emphasizes creating representations —e.g., task models, dialogue models, and abstract interaction models—that help designers understand and explore the problem space. In this approach, models are often evaluated on how well they capture user tasks and domain constraints (*accuracy*), how clearly they express interaction semantics (*understandability*), and whether they abstract away irrelevant implementation detail (*abstraction*). Because the models are usually used for analysis and communication rather than execution, they may not directly drive generation of code or platform-specific artifacts; their utility is measured more in terms of supporting design reasoning and early evaluation rather than automation. This aligns with Selic's emphasis that quality models should be expressive enough and purposeful for design, even if not directly executable. A model can exist only for reasoning about it, not necessarily to generate code.

In contrast, in MDE-UI, abstraction levels are explicitly organized—e.g., T&D, AUI, CUI, FUI—but expressiveness depends on the capabilities of the chosen UIDL or Domain-Specific Language (DSL). MDE-UI treats models as primary engineering artifacts that systematically transform across abstraction levels—e.g., from T&D to AUI, to CUI, and to FUI in forward engineering. MDE-UI optimizes automation, so that high-level models not only support understanding but predictively generate multiple variants at any level, such as platform-specific interfaces with reduced manual coding.

Consequently, the criteria extend beyond the qualities of static models: automation support and toolchain maturity become central, as does the ability to trace transformations and maintain consistency across target platforms. In MDE, models must not only be accurate and understandable, but also sufficiently precise to support automatic refinement and effective at reducing manual effort (*cost-effectiveness*). By integrating formal transformation rules and model repositories, MDE-UI strives to turn abstract interaction specifications into executable UIs systematically.

UML*4*UI approaches expressiveness differently: it relies on UML as a general-purpose modeling language capable of representing domain structure, behavioral flows, interaction logic, and presentation aspects within a unified metamodeling space. While UML may not provide UI-specific constructs out of the box, its extensibility mechanisms allow systematic coverage of CRF levels without changing modeling paradigms. This aligns with Selic's view of abstraction as a unifying engineering mechanism rather than a collection of domain-specific languages (DSLs).

6.1.2 Traceability

According to Selic, one of the core benefits of elevating models to first-class artifacts is the ability to establish explicit and systematic relationships across development stages.

In traditional MB-UID, traceability between task, abstract, and concrete levels is often conceptual rather than formally enforced: the links are typically maintained manually and primarily support reasoning, documentation, validation, and evaluation—e.g., automatic evaluation of usability guidelines on a Concrete UI vs. FUI [9], since UX is key for MDE [3]. Instead, MDE-UI creates and maintains relationships expressing mappings across the various levels, once transformations have been applied. In this way, it is possible to re-trace the steps performed for any development path and to play it again. Each transformation step can maintain explicit mappings between source and target elements, enabling impact analysis, consistency checking, round-trip engineering, and systematic evolution across platforms. The big win is: **when models change, the UI changes accordingly**.

UML*4*UI strengthens traceability by maintaining all abstraction levels within a coherent UML-based model

space. Because transformations connect elements defined within the same underlying metamodel, trace links can be explicitly represented and maintained. This supports lifecycle integration, a key concern in Selic's engineering view of MDE.

6.1.3 Transformation Effort

Selic highlights that the engineering value of models depends on their ability to drive systematic transformations, not merely to document design intent. However, defining and maintaining transformations introduces effort that must be justified by automation gains. MB-UID often relies on manual refinement, resulting in lower transformation effort but limited automation. MDE always formalizes M2M and M2T transformations, increasing initial effort while enabling repeatability.

UML*4*UI follows the same model-driven principle that transformations are central engineering artifacts. By leveraging standard UML metamodels and established transformation infrastructures, UML*4*UI may facilitate the definition and integration of transformations by providing a common modeling foundation and reducing reliance on entirely *ad hoc* solutions. Instead, the focus shifts toward defining systematic mappings across CRF levels, consistent with Selic's notion that engineering discipline requires explicit transformation semantics. However, many UI transformations remain inherently application-dependent, and the extent to which a common modeling foundation translates into reduced development effort or productivity gains requires empirical evaluation and may depend on factors such as transformation complexity, tooling support, and application domain.

Moreover, because UML*4*UI extends the modeling space without modifying the core structure of existing system models, previously defined transformations related to other software concerns can be preserved. Only those transformations that directly involve the newly introduced UI concepts need to be defined or adapted, allowing existing transformation chains to remain largely unchanged.

6.1.4 Reusability

In Selic's view [61,63], abstraction and separation of concerns are prerequisites for reuse. Models should encapsulate platform-independent intent, enabling multiple realizations. In MBUI, reuse often occurs at the task or abstract level but may require redesign when targeting different platforms. MDUI enhances reuse through abstraction layers and transformation chains.

UML*4*UI promotes reuse by clearly separating interaction intent (T&D and AUI) from realization (CUI and FUI), while maintaining conceptual continuity within UML. The running example demonstrates reuse of task and domain models across graphical, 3D, and vocal UI variants. This reflects Selic's argument that reusable models must represent essential system properties independently of platform contingencies. This is also backed up with empîrical evidence that UI reusability is supported by MBUID [22].

6.1.5 Tool Support

Selic repeatedly emphasizes that engineering credibility requires tool support and industrial viability. A modeling approach that is not connected to mature tooling is unlikely to deliver its promised benefits.

Specialized UIDLs may provide strong UI-specific tooling, but they often remain only loosely integrated with the broader software engineering environment. By contrast, UML*4*UI can build on mature UML toolchains, model repositories, transformation engines, and integrations with development-environments.

Consequently, UML*4*UI is consistent with Selic's engineering perspective: models should participate in the broader lifecycle ecosystem rather than remain isolated artifacts.

The importance of tool support is also evident in the current industrial landscape. Many low-code and no-code platforms rely heavily on model-based abstractions and visual modeling environments to enable the specification of applications, including the definition of user interfaces through visual models and platform-specific UI components. Platforms such as Mendix [39], OutSystems [48], Appian [5], and Microsoft Power Apps [41] allow developers to describe the structure, behavior, and user interface of applications through graphical models that are subsequently transformed into executable systems. In this sense, these platforms illustrate how model-driven principles, particularly model-based UI specification and the automated generation of executable artifacts, can be applied successfully at industrial scale when supported by integrated tooling.

However, such environments are typically designed for specific development domains and are often embedded in proprietary technological ecosystems and domain-specific languages. In contrast, UML*4*UI aims to leverage established standards and widely understood modeling notations. Although providing adequate tool support remains a challenge, grounding the approach in standard UML concepts may facilitate integration with existing modeling infrastructures and help mitigate the

technological fragmentation often associated with proprietary model-driven platforms.

6.1.6 Change Impact

MDE, as described by Selic [56,57], aims to reduce the cost of change by centralizing system intent in models and propagating modifications through systematic transformations.

In MB-UID without automation, changes may require manual synchronization across abstraction levels. However, in model-driven UI development changes propagate through transformation chains, but their scope depends on model coupling. UML*4*UI structures change propagation across CRF levels: modifications at the task or domain level affect derived AUI and CUI artifacts, while localized changes—e.g., widget substitution—can remain confined to concrete levels. This controlled propagation reflects Selic's argument that disciplined abstraction [62] reduces change amplification.

6.1.7 Analyzability

Finally, Selic stresses that models should support reasoning, validation, and verification. Without analyzability, models remain descriptive rather than engineering artifacts. Some UIDLs provide domain-specific analyses [42] but may lack standardized reasoning frameworks. UML models benefit from established mechanisms for structural consistency checking, behavioral reasoning, and constraint specification—e.g., via OCL or formalized transformation semantics. By keeping all abstraction levels within a unified language, UML*4*UI facilitates cross-level reasoning and consistency analysis, reinforcing its engineering legitimacy.

6.1.8 Discussion

Viewed through Selic's engineering perspective, the comparison reveals that UML*4*UI does not merely replicate existing model-based or model-driven UI approaches. Rather, it operationalizes core MDE principles—e.g., abstraction, systematic transformation, lifecycle integration, and analyzability—within the specific context of user interface engineering.

While specialized UIDLs may offer stronger UI-specific constructs, UML*4*UI prioritizes integration, traceability, and engineering discipline across the development lifecycle. In doing so, it positions user interface modeling within the broader paradigm of model-driven software engineering, rather than as a peripheral or isolated activity.

6.2 Lessons Learned and Implications

The comparative analysis highlights several insights regarding the role of modeling languages in UI engineering and the relationship between model-based and model-driven approaches. While both paradigms share the objective of elevating user interface artifacts to first-class models, their practical effectiveness strongly depends on the modeling foundations used to represent UI concerns across abstraction levels.

A first lesson concerns the importance of **expressiveness across modeling levels**. UI engineering inherently spans multiple abstractions—from tasks and domain concepts to interaction structures and platform-specific interfaces. Approaches that rely on specialized languages often provide rich constructs for specific levels of the CRF, but they may lack seamless integration with models representing other system concerns. The UML*4*UI perspective suggests that a general-purpose modeling language can provide sufficient expressiveness to represent these abstractions while maintaining conceptual continuity across levels.

A second lesson relates to **traceability and model integration**. UI models rarely exist in isolation: they are closely linked to domain models, interaction logic, and architectural elements. The comparison indicates that traceability becomes easier to maintain when UI representations share a common modeling language with other system artifacts. By leveraging UML as a unifying modeling backbone, UML*4*UI facilitates explicit relationships between UI models and the broader system model, thereby supporting consistency and co-evolution.

A third lesson concerns the **engineering effort associated with transformations and automation**. Model-driven UI approaches emphasize transformation chains that generate executable interfaces from higher-level models. While this automation can improve productivity, it also introduces complexity in the definition, maintenance, and debugging of transformation rules. Using UML as a modeling foundation does not eliminate the need for transformations, but it can reduce the conceptual gap between models and implementation artifacts by relying on well-understood modeling constructs and existing transformation infrastructures.

The fourth lesson highlights the relevance of **tooling and ecosystem support.** Dedicated UI modeling languages often require specialized tools that may not integrate easily with mainstream development environments. In contrast, UML benefits from an ecosystem of modeling tools, analysis frameworks, and transformation engines. This ecosystem can lower the adoption barrier and enable UI models to participate more naturally in model-driven development workflows. As tool

availability is not sufficient to ensure practical adoption, the use of well-established standard notations plays an important role. When modeling approaches rely on proprietary languages or platform-specific concepts, they may create barriers to adoption and limit interoperability across tools and development environments. Familiarity with notations can reduce the learning effort required to adopt new modeling approaches and facilitate their integration into existing development practices. From this perspective, approaches such as UML*4*UI may benefit not only from the availability of mature tooling but also from the use of a common modeling foundation.

Finally, the comparison reveals broader implications for the **integration of UI modeling within MDE practice**. Rather than treating UI modeling as a separate discipline with its own languages and toolchains, UML*4*UI illustrates how UI concerns can be embedded within the same modeling framework used for other system aspects. This integration supports a more coherent view of the system and aligns with Selic's vision of models as central engineering artifacts that enable abstraction, automation, and lifecycle integration.

Taken together, these lessons suggest that the choice of a modeling foundation plays a critical role in determining how effectively UI models can support model-driven engineering. By positioning UML as a unifying modeling language across CRF levels, UML*4*UI offers a pathway toward greater integration between UI engineering and the broader MDE ecosystem.

7 Related work

User interface modeling has been addressed from multiple perspectives, including domain-specific UI Description Languages (UIDLs) [42], model-based and model-driven UI engineering frameworks [25, 51], and more recently intelligent [2] and adaptive UI approaches, such as those based on reinforcement learning [30]. The main goal is to enrich MDE with AI assistance [52] provided that such an AI module would capture enough the semantics of various models and transformations. While these works share the objective of elevating UI artifacts to first-class modeling constructs, they differ in the modeling languages they adopt, the abstraction mechanisms they provide, the transformation strategies they emply, and their degree of integration with mainstream software engineering practices. In this section, we briefly review representative approaches and position UML*4*UI with respect to these research streams.

7.1 UI Description Languages (UIDLs)

A significant body of work in model-based UI engineering has focused on the definition of dedicated UI Description Languages (UIDLs), such as in alphabetic order: IFML [13], MariaXML [49], OpenUIDL [42], PlastiXML [20], SMUIML [23], UsiXML [37], UIML [33], WSXL [17], XIML [24]. These languages typically provide constructs tailored to UI abstractions across CRF levels, including task models, abstract interaction objects, concrete widgets, and navigation structures. Their primary strength lies in their domain specificity: they offer rich UI-centric abstractions and often support automated transformations toward executable user interfaces.

Recent research on model-based intelligent UI adaptation further extends this line of work by proposing conceptual frameworks to analyze and compare UI adaptation approaches [2]. In particular, a property-based reference framework has been introduced to characterize intelligent UI adaptation along dimensions such as adaptation responsibility, automation degree, adaptation logic, CRF level, and tool support [2]. This work highlights that the proliferation of UI adaptation approaches makes systematic comparison difficult and calls for structured evaluation criteria to position methods in the state of the art [2].

While UIDLs and adaptation frameworks provide UI-specific expressiveness and structured comparison dimensions, they often remain disconnected from mainstream software engineering modeling ecosystems. Their tooling, metamodeling infrastructures, and transformation chains are frequently specialized and not directly integrated with general-purpose modeling environments.

In contrast, UML*4*UI does not introduce a new UIDL. Instead, it investigates how UML, through its extensibility mechanisms and integration with MDE toolchains, can serve as a foundational modeling language capable of representing UI abstractions across CRF levels while preserving alignment with broader software engineering artifacts.

7.2 General-purpose vs Domain-specific Modeling

It is well established in the MDE literature that there is a persistent tension between general-purpose modeling languages (GPLs) and domain-specific languages (DSLs). DSLs, including many UIDLs, typically provide higher domain expressiveness and more concise representations for UI concerns. By embedding UI semantics directly into the language, these approaches can facilitate automation, adaptation, and analysis at specific levels of abstraction. As a result, DSL-based solutions

often enable efficient modeling and generation mechanisms tailored to particular UI technologies or interaction paradigms.

However, DSL-based approaches may also introduce fragmentation when multiple modeling languages are used across different system concerns—e.g., domain, business logic, architecture, and UI. In such cases, different languages, metamodels, and toolchains must coexist within the same development process. This fragmentation can complicate traceability across models, increase the effort required for model management and co-evolution, and make integration with lifecycle tooling more difficult. As systems evolve, maintaining consistency across heterogeneous modeling artifacts may therefore become a significant engineering challenge.

Recent work on model-based intelligent UI adaptation underscores this challenge by identifying traceability across heterogeneous models and co-evolution between UI and underlying software components as open research issues [2]. These concerns resonate strongly with broader MDE challenges related to model management, synchronization, and runtime models.

UML*4*UI explores an alternative direction: rather than relying on a dedicated UI DSL, it leverages a widely adopted general-purpose modeling language as a unifying backbone. UML's structural, behavioral, and interaction diagrams, combined with stereotypes and profiles, enable the representation of UI concerns without leaving the mainstream modeling ecosystem. This approach trades some domain-specific conciseness for improved integration, lifecycle coherence, and cross-level traceability.

7.3 Positioning of UML*4*UI

UML*4*UI can be positioned at the intersection of model-based UI engineering and model-driven software engineering. Like UIDL-based approaches, it proposes multi-level abstractions aligned with the CRF and supports forward, reverse, and lateral engineering paths. However, its distinguishing characteristic lies in its commitment to using UML as a foundational modeling language across all abstraction levels.

Compared to property-based frameworks for intelligent UI adaptation [2], UML*4*UI is orthogonal in scope. Adaptation frameworks focus primarily on characterizing who adapts, what is adapted, how adaptation is triggered, and at which CRF level adaptation occurs. UML*4*UI, in contrast, emphasizes the engineering properties of the modeling backbone itself: expressiveness across levels, traceability, transformation effort, reusability, tool support, change impact, and analyzability. These criteria assess whether a modeling approach operationalizes core MDE principles, particularly those articulated by Selic regarding abstraction, systematic transformation, and lifecycle integration.

Thus, UML*4*UI does not compete with domain-specific UIDLs or adaptation-centric frameworks; rather, it complements them by providing a general-purpose modeling infrastructure capable of integrating UI concerns with broader system models. In doing so, it contributes to bridging the gap between specialized UI modeling research and mainstream model-driven engineering practice.

8 Final Remarks and Open Research Challenges

This paper explored how UML can serve as a unifying modeling foundation for UI engineering across the abstraction levels defined in the CRF. By introducing UML*4*UI, we investigated how a widely adopted general-purpose modeling language can be leveraged to represent UI concerns across task, abstract, concrete, and final interface levels. In particular, UML*4*UI shows how existing UML constructs can support the representation of presentation, interaction, and behavioral aspects of user interfaces while preserving traceability across CRF levels.

At the same time, the discussion throughout the paper highlights that significant challenges remain before such an approach can fully support the engineering of modern UIs. Several research directions therefore emerge as promising topics for further investigation.

- **Formalization of UI modeling semantics within UML-based approaches.** While UML provides expressive constructs for representing structure and behavior, the precise semantics of UI-specific concepts—e.g., presentation elements, interaction objects, navigation flows— often remain implicit. Future work should explore how UML profiles, semantic annotations, or metamodel extensions could provide a more rigorous definition of these UI abstractions while remaining compatible with standard UML tooling and modeling practices.
- **Systematic transformations across CRF levels.** Although transformation techniques are well established in MDE, defining robust and reusable transformations between task models, abstract interaction models, and concrete UI representations remains challenging. In particular, ensuring traceability across levels, supporting bidirectional transformations, and preserving interaction semantics are important topics for further research.

- **Co-evolution and traceability between UI models and other system models.** User interfaces rarely evolve independently from the rest of the system. Changes in domain models, services, or architectural components usually require corresponding changes in the interface. A unified modeling backbone such as UML*4*UI opens the possibility of explicitly linking UI artifacts with domain, behavioral, and architectural models, but effective mechanisms for maintaining these relationships throughout system evolution remain an open issue.
- **Tooling support and integration with development workflows.** While UML benefits from an ecosystem of modeling tools, further work is needed to support UI-specific modeling constructs, automated transformations, and seamless integration with implementation technologies. Improving the usability of modeling tools and aligning them with the practices of UI designers and developers will be essential for bridging the gap between modeling research and industrial adoption.
- **Data-driven and intelligent user interfaces.** Machine learning techniques are increasingly used to personalize interfaces, recommend actions, or adapt interaction flows based on user behavior. Integrating such data-driven mechanisms with model-based and model-driven UI representations remains challenging, particularly in terms of transparency, explainability, and maintaining the engineering discipline traditionally associated with MDE.
- **Conversational and generative modeling of UML-based UI specifications.** Recent advances in generative AI open new possibilities for conversational modeling environments that raise the level of abstraction at which UI requirements can be specified. Developers could interact with modeling tools using natural language to describe user tasks, interaction scenarios, or UI structures, while the system generates corresponding UML artifacts aligned with the CRF levels (e.g., task models, interaction diagrams, or UI component structures). Such approaches could lower the entry barrier to model-driven UI engineering while preserving the rigor and traceability provided by UML-based models.

Summarizing, this work suggests that positioning UML as a unifying modeling foundation for UI engineering can contribute to bridging the gap between specialized UI modeling approaches and mainstream model-driven software engineering. By aligning UI modeling with the broader MDE foundations, UML*4*UI offers a conceptual framework toward more integrated and disciplined engineering of UIs. At the same time, addressing the challenges outlined above will be essential to fully realize the potential of this perspective and to support the next generation of model-driven user interface engineering.

Acknowledgements This work is supported by the UCI-Adapt project (PID2022-140106NB-I00) funded by the State Research Agency (AEI). Jean Vanderdonckt is supported by the EU EIC Pathfinder-Awareness Inside challenge "Symbiotik" project (1 Oct. 2022–30 Sep. 2026) under Grant no. 101071147.

References

1. D1.3 - UsiXML Definition. Tech. rep. (2012). URL: https://itea4.org/project/workpackage/document/download/1583/D1.3.%20UsiXML%20-%20Definition.pdf
2. Abrahão, S., Insfrán, E., Sluÿters, A., Vanderdonckt, J.: Model-based intelligent user interface adaptation: challenges and future directions. Journal on Software and Systems Modelling **20**(5), 1335–1349 (2021). DOI 10.1007/S10270-021-00909-7
3. Abrahão, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., Whittle, J.: User experience for model-driven engineering: Challenges and future directions. In: Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems, MODELS '17, pp. 229–236. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/MODELS.2017.5
4. Akiki, P.A., Bandara, A.K., Yu, Y.: Engineering adaptive model-driven user interfaces. IEEE Transactions on Software Engineering **42**(12), 1118–1147 (2016). DOI 10.1109/TSE.2016.2553035
5. Appian: Appian low-code platform. https://appian.com/. Accessed: 2026-03-08
6. Aquino, N., Vanderdonckt, J., Panach, J.I., Pastor, O.: Conceptual modelling of interaction. In: D.W. Embley, B. Thalheim (eds.) Handbook of Conceptual Modeling - Theory, Practice, and Research Challenges, pp. 335–358. Springer (2011). DOI 10.1007/978-3-642-15865-0_10
7. Aquino, N., Vanderdonckt, J., Pastor, O.: Transformation templates: Adding flexibility to model-driven engineering of user interfaces. In: Proceedings of the ACM International Symposium on Applied Computing, SAC '10, pp. 1195–1202. ACM, New York, NY, USA (2010). DOI 10.1145/1774088.1774340
8. Aquino, N., Vanderdonckt, J., Valverde, F., Pastor, O.: Using profiles to support model transformations in the model-driven development of user interfaces. In: V. López-Jaquero, F.M. Simarro, J.P.M. Massó, J. Vanderdonckt (eds.) Computer-Aided Design of User Interfaces VI, Proceedings of the Seventh International Conference on Computer-Aided Design of User Interfaces, CADUI '08, pp. 35–46. Springer (2008). DOI 10.1007/978-1-84882-206-1_4
9. Beirekdar, A., Vanderdonckt, J., Noirhomme-Fraiture, M.: A Framework and a Language for Usability Automatic Evaluation of Web Sites by Static Analysis of HTML Source Code, pp. 337–348. CADUI '02. Springer Netherlands, Dordrecht (2002). DOI 10.1007/978-94-010-0421-3_29
10. Blair, G., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer **42**(10), 22–27 (2009). DOI 10.1109/MC.2009.326
11. Bouillon, L., Vanderdonckt, J.: Retargeting web pages to other computing platforms with vaquita. In: Ninth Working Conference on Reverse Engineering, 2002. Proceedings., pp.

339–348 (2002). DOI 10.1109/WCRE.2002.1173091. URL https://doi.org/10.1109/WCRE.2002.1173091
12. Bouillon, L., Vanderdonckt, J., Eisenstein, J.: Model-based approaches to reengineering web pages. In: C. Pribeanu, J. Vanderdonckt (eds.) Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design, TAMODIA '02, pp. 86–95. INFOREC Publishing House Bucharest (2002). DOI 10.5555/646617.697239
13. Brambilla, M., Mauri, A., Umuhoza, E.: Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end. In: I. Awan, M. Younas, X. Franch, C. Quer (eds.) Mobile Web Information Systems, pp. 176–191. Springer International Publishing, Cham (2014)
14. Brel, C., Renevier-Gonin, P., Giboin, A., Riveill, M., Dery, A.: Reusing and combining ui, task and software component models to compose new applications. In: D. Fitton, M. Horton, J.C. Read, G. Sim (eds.) Proceedings of the 28th International BCS Human-Computer Interaction Conference, HCI '04. British Computer Society, Oxford (2014). URL https://www.scienceopen.com/document?vid=d857ccd9-4660-4627-9d71-ee2db9d1d28a
15. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Software and System Modeling **19**(1), 5–13 (2020). DOI 10.1007/s10270-019-00773-6
16. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interacting with Computers **15**(3), 289–308 (2003). DOI 10.1016/S0953-5438(03)00010-9
17. Chamberlain, D., Díaz, A., Gisolfi, D., Konuru, R.B., Lucassen, J.M., MacNaught, J., Maes, S.H., Merrick, R., Mundel, D., Raman, T.V., Ramaswamy, S., Schaeck, T., Thompson, R., Wiecha, C.: WSXL: A web services language for integrating end-user experience. In: C. Kolski, J. Vanderdonckt (eds.) Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces, Computer-Aided Design of User Interfaces III, pp. 35–50. Kluwer, Dordrecht (2002). DOI 10.1007/978-94-010-0421-3_3
18. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of UML models: a systematic review of research and practice. Softw. Syst. Model. **18**(3), 2313–2360 (2019). DOI 10.1007/S10270-018-0675-4
19. Clark, T., France, R.B., Gogolla, M., Selic, B.: Metamodeling model-based engineering tools (dagstuhl seminar 13182). Dagstuhl Reports **3**(4), 188–226 (2013). DOI 10.4230/DAGREP.3.4.188
20. Collignon, B., Vanderdonckt, J., Calvary, G.: An intelligent editor for multi-presentation user interfaces. In: Proceedings of the ACM Symposium on Applied Computing, SAC '08, pp. 1634–1641. ACM, New York, NY, USA (2008). DOI 10.1145/1363686.1364072
21. Conallen, J.: Building Web Applications with UML. Addison-Wesley (2003)
22. Delgado, A., Estepa, A., Troyano, J., Estepa, R.M.: Reusing ui elements with model-based user interface development. International Journal of Human-Computer Studies **86**, 48–62 (2016). DOI https://doi.org/10.1016/j.ijhcs.2015.09.003
23. Dumas, B., Lalanne, D., Ingold, R.: Description languages for multimodal interaction: a set of guidelines and its illustration with smuiml. Journal on Multimodal User Interfaces **3**(3), 237–247 (2010). DOI 10.1007/s12193-010-0043-3
24. Eisenstein, J., Vanderdonckt, J., Vanderdonckt, J., Puerta, A.: Adapting to mobile contexts with user-interface modeling. In: Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications, pp. 83–92. Institute of Electrical and Electronics Engineers, Los Alamitos, CA, USA (2000). DOI 10.1109/MCSA.2000.895384
25. Erazo-Garzón, L., Suquisupa, S., Bermeo, A., Cedillo, P.: Model-driven engineering applied to user interfaces. a systematic literature review. In: M. Botto-Tobar, M. Zambrano Vizuete, S. Montes León, P. Torres-Carrión, B. Durakovic (eds.) Applied Technologies, pp. 575–591. Springer Nature Switzerland, Cham (2023). DOI 10.1007/978-3-031-24985-3_42
26. Florins, M., Simarro, F.M., Vanderdonckt, J., Michotte, B.: Splitting rules for graceful degradation of user interfaces. In: Proceedings of the Working Conference on Advanced Visual Interfaces, AVI '06, p. 59–66. Association for Computing Machinery, New York, NY, USA (2006). DOI 10.1145/1133265.1133276
27. Fonseca, J.M.C.: Model-based ui xg final report (2010). URL https://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/. Contributors: Juan M. Gonzalez Calleros, Gerrit Meixner, Fabio Paterno, Jaroslav Pullmann, Dave Raggett, Daniel Schwabe, Jean Vanderdonckt
28. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE '07), pp. 37–54 (2007). DOI 10.1109/FOSE.2007.14
29. Furtado, E., Furtado, V., Silva, W.B., Rodrigues, D.W.T., da Silva Taddeo, L., Limbourg, Q., Vanderdonckt, J.: An ontology-based method for designing multiple user interfaces. In: Proceedings of Int. Workshop on Multiple User Interfaces, MUI' 01 (2001). URL https://www.researchgate.net/publication/2567741_An_Ontology-Based_Method_for_Universal_Design_of_User_Interfaces
30. Gaspar-Figueiredo, D., Vanderdonckt, J., Abrahão, S., Insfrán, E.: User experience with adaptive user interfaces: Comparing performance and preferences. J. Syst. Softw. **231**, 112,598 (2026). DOI 10.1016/J.JSS.2025.112598
31. Genaro Motti, V., Raggett, D., Van Cauwelaert, S., Vanderdonckt, J.: Simplifying the development of cross-platform web user interfaces by collaborative model-based design. In: Proceedings of the 31st ACM International Conference on Design of Communication, SIGDOC '13, pp. 55–64. ACM, New York, NY, USA (2013). DOI 10.1145/2507065.2507067
32. Gómez, J., Cachero, C.: OO-H method: extending UML to model web interfaces. In: Information modeling for internet applications, pp. 144–173. IGI Global Scientific Publishing (2003)
33. Helms, J., Schaefer, R., Luyten, K., Vermeulen, J., Abrams, M., Coyette, A., Vanderdonckt, J.: Human-Centered Engineering Of Interactive Systems With The User Interface Markup Language, pp. 139–171. Springer, London (2009). DOI 10.1007/978-1-84800-907-3_7
34. ISO: ISO/IEC 25010 - Software Quality Product Standard. standard, International Standard Organization, Geneva (2019). URL https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0
35. Koch, N., Kraus, A.: The expressive power of uml-based web engineering. In: Second Int. Worskhop on Web-oriented Software Technology (IWWOST 2002), Málaga, Spain (2002)
36. Limbourg, Q., Vanderdonckt, J.: Multipath Transformational Development of User Interfaces with Graph Trans-

formations, pp. 107–138. Springer London, London (2009). DOI 10.1007/978-1-84800-907-3_6
37. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Florins, M.: USIXML: A user interface description language supporting multiple levels of independence. In: M. Matera, S. Comai (eds.) Proceedings of Workshops in connection with the 4th International Conference on Web Engineering (ICWE '04). Engineering Advanced Web Applications, DIWE '04, pp. 325–338. Rinton Press (2004)
38. Meixner, G., Paternò, F., Vanderdonckt, J.: Past, present, and future of model-based user interface development. i-com Zeitschrift für interaktive und kooperative Medien **10**(3), 2–11 (2011). DOI https://doi.org/10.1524/icom.2011.0026
39. Mendix: Mendix low-code platform. `https://www.mendix.com/`. Accessed: 2026-03-08
40. Michotte, B., Vanderdonckt, J.: Grafixml, a multi-target user interface builder based on usixml. In: Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08), pp. 15–22 (2008). DOI 10.1109/ICAS.2008.29
41. Microsoft Power Apps: Microsoft Power Apps. `https://www.microsoft.com/en-gb/power-platform/products/power-apps`. Accessed: 2026-03-08
42. Moldovan, A., Nicula, V., Pasca, I., Popa, M., Namburu, J.K., Oros, A., Brie, P.: OpenUIDL, a user interface description language for runtime omni-channel user interfaces. Proc. ACM Hum.-Comput. Interact. **4**(EICS) (2020). DOI 10.1145/3397874
43. Montero, F., López-Jaquero, V., Vanderdonckt, J., González, P., Lozano, M., Limbourg, Q.: Solving the mapping problem in user interface design by seamless integration in idealxml. In: S.W. Gilroy, M.D. Harrison (eds.) Interactive Systems. Design, Specification, and Verification, pp. 161–172. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
44. Müller, I., Selic, B.: Stakeholders: Going beyond just "end users". IEEE Software **39**(1), 112–113 (2022). DOI 10.1109/MS.2021.3120177
45. Object Management Group: Interaction flow modeling language (ifml) specification (2014)
46. Object Management Group: OMG Unified Modeling Language (OMG UML). Tech. rep., Object Management Group (2017). URL `https://www.omg.org/spec/UML/2.5.1/`
47. OMG: Model Driven Architecture (MDA) Guide rev. 2.0. Tech. rep., Object Management Group (2014). URL: `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01`
48. OutSystems: Outsystems low-code platform. `https://www.outsystems.com/`. Accessed: 2026-03-08
49. Paternò, F., Santoro, C., Spano, L.D.: MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput. Hum. Interact. **16**(4), 19:1–19:30 (2009). DOI 10.1145/1614390.1614394
50. Ramón, O.S., Vanderdonckt, J., Molina, J.G.: Re-engineering graphical user interfaces from their resource files with usiresourcer. In: IEEE 7th International Conference on Research Challenges in Information Science (RCIS), pp. 1–12 (2013). DOI 10.1109/RCIS.2013.6577696
51. Ruiz, J., Serral, E., Snoeck, M.: Evaluating user interface generation approaches: model-based versus model-driven development. Software & Systems Modeling pp. 2753—-2776 (2018). DOI 10.1007/s10270-018-0698-x
52. Savary-Leblanc, M.: Improving mbse tools ux with ai-empowered software assistants. In: Proceedings of ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C'19, pp. 648–652. IEEE Computer Society, Los Alamitos, CA, USA (2019). DOI 10.1109/MODELS-C.2019.00099
53. Schlee, M., Vanderdonckt, J.: Generative programming of graphical user interfaces. In: Proceedings of the Working Conference on Advanced Visual Interfaces, AVI '04, p. 403–406. Association for Computing Machinery, New York, NY, USA (2004). DOI 10.1145/989863.989936
54. Selic, B.: Using uml for modeling complex real-time systems. In: F. Mueller, A. Bestavros (eds.) Proceedings of the ACM SIGPLAN International Workshop on Languages, Compilers, and Tools for Embedded Systems, LCTES '98, Montreal, Canada, June 19-20, 1998, Lecture Notes in Computer Science, pp. 250–260. Springer, Berlin, Heidelberg (1998). DOI 10.1007/BFb0057795
55. Selic, B.: From model-driven development to model-driven engineering. Proceedings of the 2nd International Workshop on Model-Driven Engineering, Verification, and Validation (2003)
56. Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5), 19–25 (2003). DOI 10.1109/MS.2003.1231146
57. Selic, B.: Model-driven development: its essence and opportunities. In: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC'06, pp. 7 pp.– (2006). DOI 10.1109/ISORC.2006.54
58. Selic, B.: UML 2: A model-driven development tool. IBM Systems Journal **45**(3), 607–620 (2006). DOI 10.1147/SJ.453.0607
59. Selic, B.: From model-driven development to model-driven engineering. In: 19th Euromicro Conference on Real-Time Systems (ECRTS'07), pp. 3–3 (2007). DOI 10.1109/ECRTS.2007.16
60. Selic, B.: A systematic approach to domain-specific language design using uml. In: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC '07, pp. 2–9 (2007). DOI 10.1109/ISORC.2007.10
61. Selic, B.: Model-based software engineering: Expected and unexpected challenges. In: A. Moreira, M.J. Suárez-Cabal, C. de la Riva, J. Tuya (eds.) XIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2008), Gijón, Spain, October 7-10, 2008. Proceedings, pp. 357–358 (2008)
62. Selic, B.: A short catalogue of abstraction patterns for model-based software engineering. Int. J. Softw. Informatics **5**(1-2), 313–334 (2011). URL `http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=i86`
63. Selic, B.: Programming $\subset$ modeling $\subset$ engineering. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 9953, pp. 11–26 (2016). DOI 10.1007/978-3-319-47169-3_2
64. Selic, B.: Modeling of real-time software systems. In: Y. Tian, D.C. Levy (eds.) Handbook of Real-Time Computing, pp. 25–98. Springer (2022). DOI 10.1007/978-981-287-251-7_57
65. Sottet, J.S., Calvary, G., Coutaz, J., Favre, J.M.: A model-driven engineering approach for the usability of plastic user interfaces. In: J. Gulliksen, M.B. Harning, P. Palanque, G.C. van der Veer, J. Wesson (eds.) Engineering Interactive Systems, pp. 140–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

66. Stanciulescu, A., Vanderdonckt, J.: Design options for multimodal web applications. In: G. Calvary, C. Pribeanu, G. Santucci, J. Vanderdonckt (eds.) Computer-Aided Design Of User Interfaces V, Proceedings of the Sixth International Conference on Computer-Aided Design of User Interfaces, CADUI 2006 6-8 June 2006, Bucharest, Romania, pp. 41–56. Springer (2006). DOI 10.1007/978-1-4020-5820-2_4
67. Tran, V., Vanderdonckt, J., Tesoriero, R., Beuvens, F.: Systematic generation of abstract user interfaces. In: S.D.J. Barbosa, J.C. Campos, R. Kazman, P.A. Palanque, M.D. Harrison, S. Reeves (eds.) Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, pp. 101–110. ACM (2012). DOI 10.1145/2305484.2305502
68. Vanderdonckt, J.: A mda-compliant environment for developing user interfaces of information systems. In: O. Pastor, J.F. e Cunha (eds.) Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3520, pp. 16–31. Springer (2005). DOI 10.1007/11431855_2
69. Vanderdonckt, J., Berquin, P.: Towards a very large model-based approach for user interface development. In: Proceedings User Interfaces to Data Intensive Systems, pp. 76–85 (1999). DOI 10.1109/UIDIS.1999.791464
70. Vanderdonckt, J., Bodart, F.: Encapsulating knowledge for intelligent automatic interaction objects selection. In: B. Arnold, G.C. van der Veer, T.N. White (eds.) Human-Computer Interaction, INTERACT '93, IFIP TC13 International Conference on Human-Computer Interaction, 24-29 April 1993, Amsterdam, The Netherlands, jointly organised with ACM Conference on Human Aspects in Computing Systems CHI'93, pp. 424–429. ACM (1993). DOI 10.1145/169059.169340
71. Vanderdonckt, J., González-Calleros, J.M.: Task-driven plasticity: One step forward with ubidraw. In: P. Forbrig, F. Paternò (eds.) Engineering Interactive Systems, Proc. of Second Conference on Human-Centered Software Engineering, HCSE 2008, and 7th International Workshop on Task Models and Diagrams, TAMODIA 2008, Pisa, Italy, September 25-26, 200, *Lecture Notes in Computer Science*, vol. 5247, pp. 181–196. Springer (2008). DOI 10.1007/978-3-540-85992-5_16
72. W3C: Concur Task Trees (CTT). Tech. rep., World Wide Web Consortium (2012). URL: `https://www.w3.org/2012/02/ctt/`
73. Wegner, P.: Why interaction is more powerful than algorithms. Commun. ACM **40**(5), 80–91 (1997). DOI 10.1145/253769.253801
74. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Software **31**(3), 79–85 (2014). DOI 10.1109/MS.2013.65
75. Winckler, M., Vanderdonckt, J., Stanciulescu, A., Trindade, F.: Cascading dialog modeling with usixml. In: T.C.N. Graham, P. Palanque (eds.) Interactive Systems. Design, Specification, and Verification, pp. 121–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
76. Yigitbas, E., Jovanovikj, I., Biermeier, K., Sauer, S., Engels, G.: Integrated model-driven development of self-adaptive user interfaces. Software and Systems Modelling **19**(5), 1057–1081 (2020). DOI 10.1007/s10270-020-00777-7

A UML metamodels Involved

A.1 Context Metamodel

The context metamodel in UsiXML defines the contextual information that influences the behavior and adaptation of interactive systems. It provides a structured representation of the environment in which user interaction occurs, including the user, the platform, and the surrounding environment. This metamodel enables the development of context-aware and adaptive user interfaces. The context metamodel typically organizes context into three main dimensions: user context, platform context, and environmental context. User context includes characteristics such as preferences, roles, skills, and disabilities. Platform context describes device capabilities, software resources, and interaction modalities. Environmental context represents external conditions such as location, lighting, noise, or network connectivity.

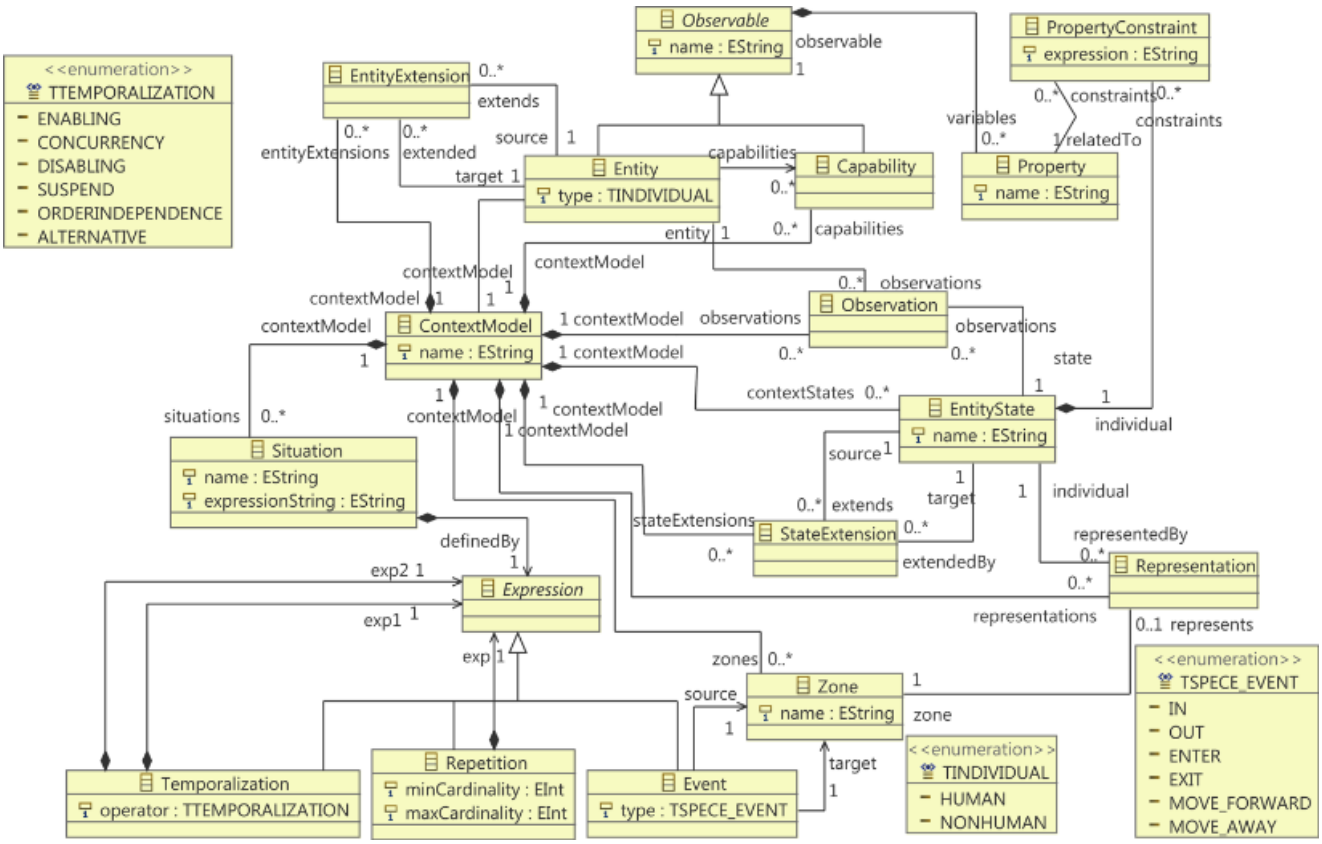


Fig. 25: Context metamodel as a UML class diagram (extracted from [1])

By modeling these contextual elements, the framework can dynamically adapt user interfaces according to changing situations and usage conditions. The context metamodel works closely with the transformation, mapping, and resource metamodels to support plasticity, usability, and multimodal interaction.

A.2 Task Metamodel

The task metamodel in UsiXML provides a formal structure for describing user and system activities within interactive applications. It belongs to the Tasks & Concepts layer of the UsiXML framework and supports model-driven user interface development. The metamodel combines hierarchical task decomposition from HTA (Hierarchical Task Analysis) with temporal relationships inspired by CTT and LOTOS operators. At its core, the metamodel is centered on the TaskModel, which organizes tasks into hierarchies composed of atomic and complex tasks. Atomic tasks represent indivisible actions, while composed tasks can be decomposed into subtasks. Each task includes descriptive attributes such as name, identifier, preconditions, postconditions, and canonical task types like CREATE, SELECT, MODIFY, or NAVIGATE.

The metamodel also defines TaskExpression and TemporalRelationship elements to describe dependencies and execution order between tasks using operators such as ENABLING, CONCURRENCY, CHOICE, and SUSPEND. To enrich task descriptions, Decorations are used to specify properties such as criticity, frequency, iteration constraints, execution time, and interaction nature (USER, SYSTEM, or INTERACTIVE).

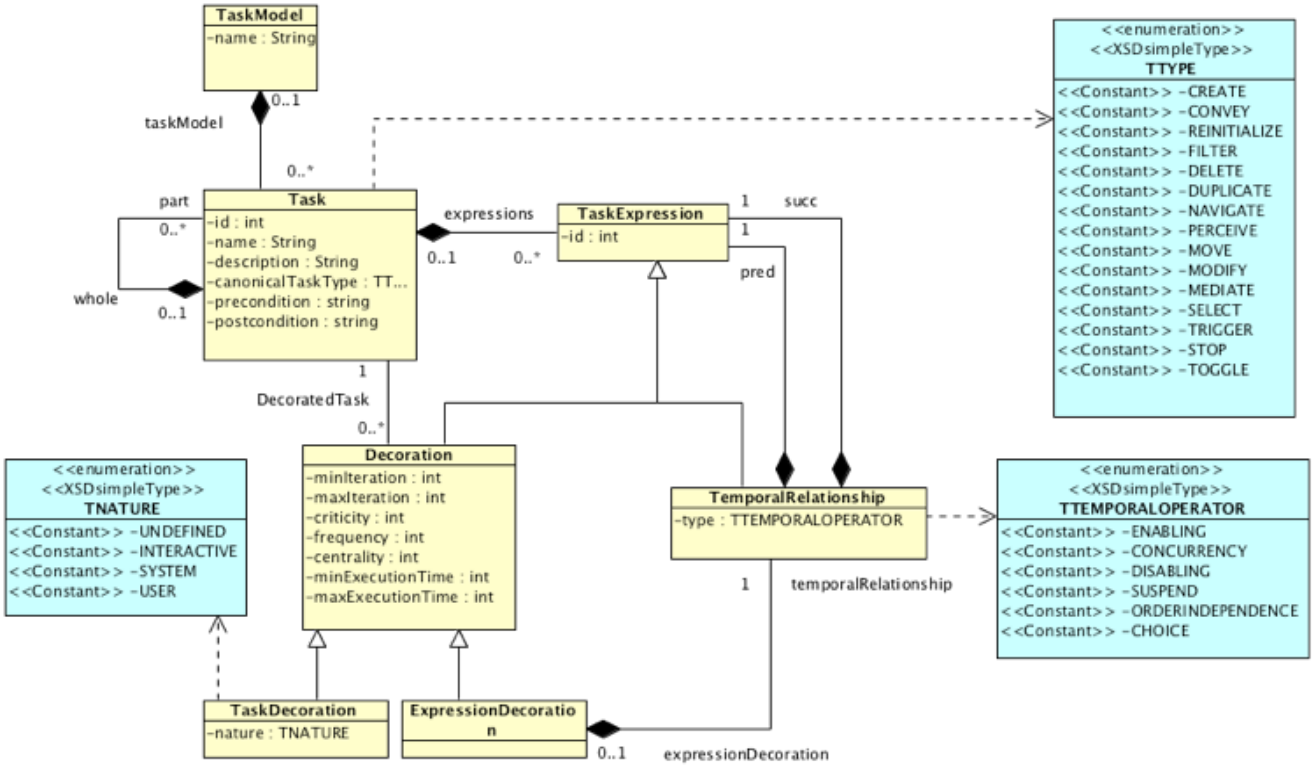


Fig. 26: Task metamodel as a UML class diagram (extracted from [1])

Overall, the task metamodel offers an extensible and structured way to model user activities, collaborative interactions, and temporal behavior, enabling the systematic design and analysis of interactive systems.

A.3 Domain Metamodel

The domain metamodel in UsiXML describes the entities manipulated by users while interacting with a system and defines the relationships among these entities. It is based on the UML class diagram metamodel and serves as the conceptual representation of application data and business logic within the UsiXML framework. The domain metamodel enables the specification of system structure independently from user interface implementation details.

At the center of the metamodel is the DomainModel, which groups together classes, attributes, operations, and relationships representing the application domain. The main concept is the Class, which models entities of the system along with their properties and behaviors. Classes may contain Attributes that define data characteristics and Operations that define available actions or functionalities.

Relationships between entities are represented through Associations and Generalizations. Associations describe semantic links between classes and may include multiplicities, navigability, or aggregation types such as shared or composite aggregation. Generalizations represent inheritance relationships, allowing subclasses to inherit characteristics from more general classes. The metamodel also supports interfaces, dependencies, realizations, and association classes to capture complex object-oriented structures. Additional concepts such as Property, Parameter, MultiplicityElement, and VisibilityKind provide detailed specifications for attributes, operations, and accessibility constraints. Enumerations define standardized values for aggregation types, parameter directions, and visibility levels.

Overall, the domain metamodel provides a structured and extensible representation of application data and object relationships. By relying on UML principles, it supports interoperability, abstraction, and clear separation between domain concepts and user interface design.

A.4 Abstract User Interface Metamodel

The Abstract User Interface (AUI) metamodel in UsiXML represents the user interface at a platform-independent level. It corresponds to the Platform-Independent Model (PIM) in Model-Driven Engineering (MDE) and describes user interactions without considering concrete widgets, devices, modalities, or implementation technologies. The purpose of the AUI metamodel is to provide a high-level description of interaction spaces and user interaction logic that can later be transformed into concrete user interfaces adapted to different platforms and contexts of use. The AUI metamodel is centered on the AbstractUIModel, which organizes the interface into a collection of abstract interaction units. These interaction units represent logical interaction elements rather than physical interface

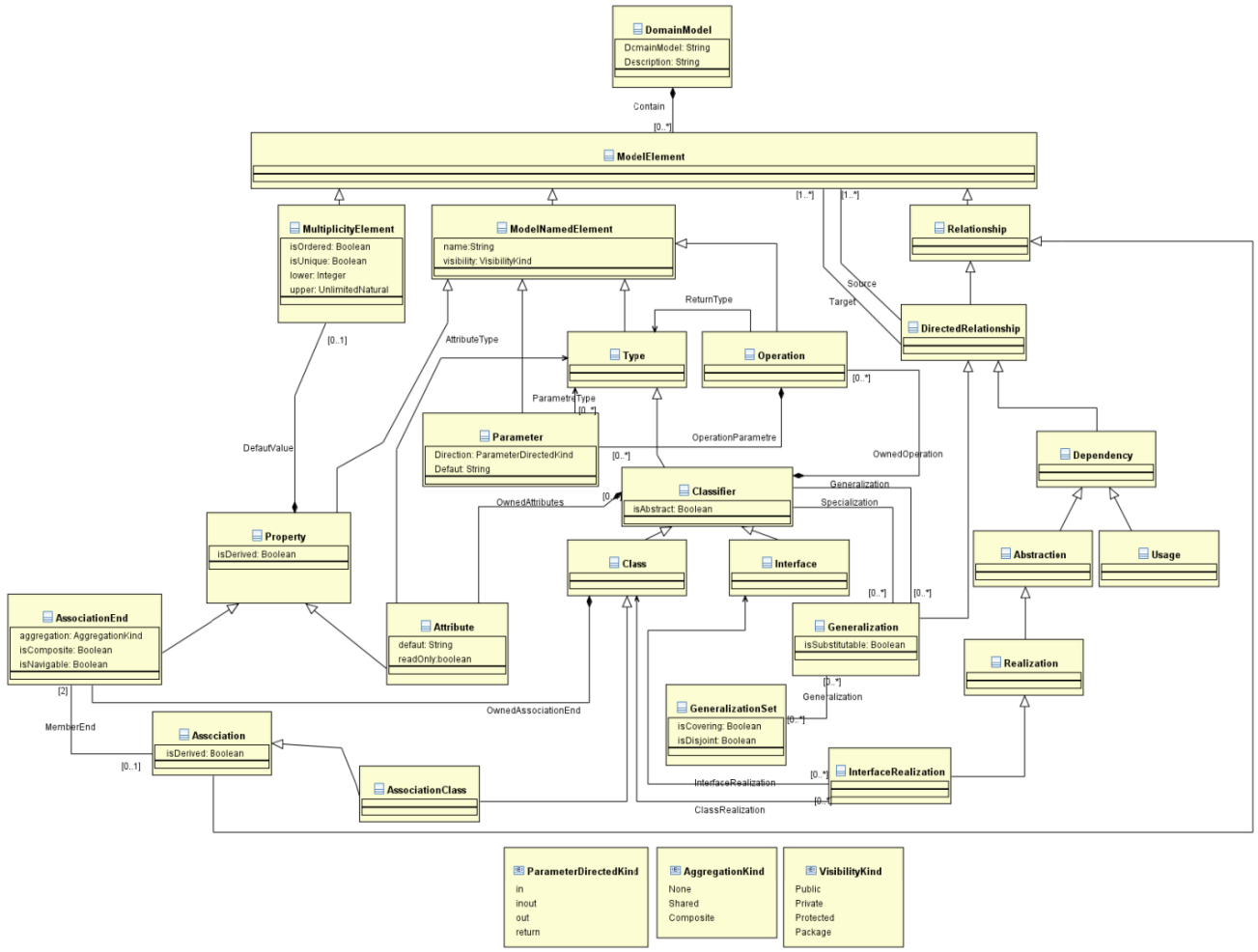


Fig. 27: Domain metamodel as a UML class diagram (extracted from [1])

components. The model is designed to be independent of graphical, vocal, tactile, or multimodal implementation details, ensuring portability and adaptability across platforms.

The core concept of the metamodel is the AbstractInteractionUnit. Every abstract object in the interface is represented by this entity. Each interaction unit contains descriptive attributes such as identifiers, roles, hierarchy order, repetition factors, and importance levels. Security aspects are also integrated through attributes such as authentication type and security mechanism. Authentication may be performed using passwords, integrated mechanisms such as certificates, or may require no authentication at all. Additional security mechanisms, such as CAPTCHA validation, can also be specified.

To support internationalization and localization, the metamodel defines the AbstractLocalization entity. This element stores textual descriptions related to interaction units, including labels, short labels, long labels, descriptions, and contextual help messages in multiple languages. This mechanism enables the same abstract interface to be adapted to different linguistic and cultural environments.

The AUI metamodel structures interaction units into compound and elementary units. AbstractCompoundIU represents composite interaction spaces that group multiple interaction units together. A specialized form, AbstractSelectionIU, models interaction mechanisms based on selecting items from a list or collection. It supports several selection behaviors such as single selection, interval selection, multiple interval selection, continuous selection, expandable lists, and rating systems. Additional attributes define ordering criteria, selection ranges, and step values for numerical selections.

Elementary interaction units are represented by AbstractElementaryIU, which is further specialized into AbstractDataIU and AbstractTriggerIU. AbstractDataIU handles interactions related to data manipulation and presentation. It defines references to the domain model, cardinality constraints, default values, data types, formats, and interaction modes such as input, output, or bidirectional interaction. The supported data types include text, numeric values, dates, multimedia content, arrays, and secret data. The metamodel also supports dynamic data whose content may evolve over time.

AbstractTriggerIU represents interaction units that initiate actions or navigation events. Trigger units can support navigation actions, operational commands, or combined operation-navigation behaviors. These units abstract the mechanisms by which users activate system functionalities independently from the final interface modality.

The behavioral aspects of the abstract interface are handled through AbstractListener entities. These listeners implement Event-Condition-Action (ECA) rules that define how interaction units react to user actions or system events. Event handling is represented using EventExpression structures composed of AtomicEvents and TemporalEventExpressions. Atomic events include activities such as

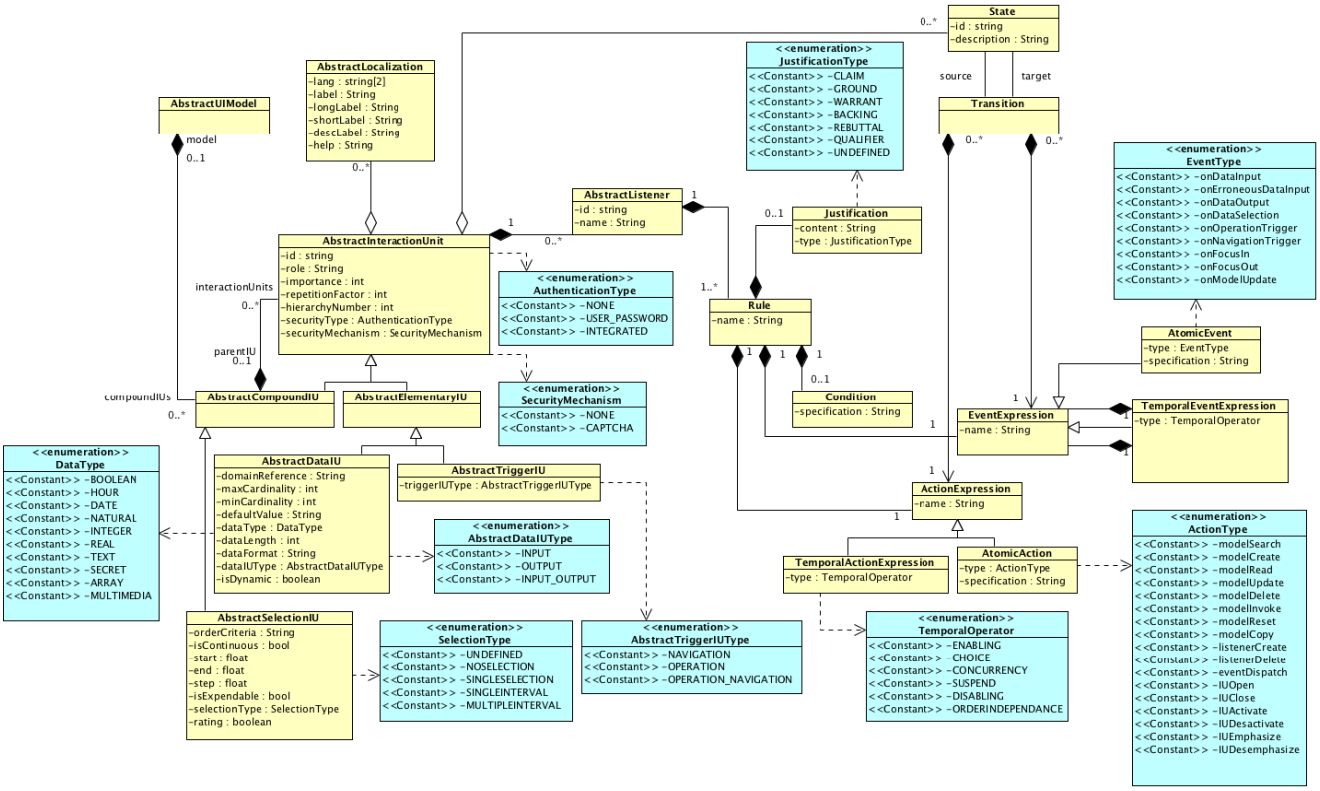


Fig. 28: Abstract User Interface metamodel as a UML class diagram (extracted from [1])

data input, erroneous input, data selection, focus changes, navigation triggers, and model updates. Temporal event expressions define relationships between events using operators such as enabling, concurrency, suspension, disabling, choice, and order independence.

Conditions specify logical expressions that must be satisfied before actions are executed. Actions themselves are represented by ActionExpression entities composed of AtomicActions and TemporalActionExpressions. Atomic actions can manipulate the domain model through operations such as create, read, update, delete, reset, or copy. They can also manage listeners, dispatch events, or control interaction units by opening, closing, activating, or emphasizing them. Temporal relationships among actions allow the specification of complex behavioral flows.

The metamodel also incorporates concepts inspired by state machines through the State and Transition entities. States represent possible conditions of an interaction unit, while transitions define movements between states triggered by events and associated actions. This integration allows compliance with W3C SCXML principles and supports dynamic, event-driven user interface behavior.

Another important aspect of the AUI metamodel is the inclusion of Justification entities. These entities document the rationale behind interface decisions and ECA rules using concepts such as claims, grounds, warrants, rebuttals, and qualifiers. Although not directly involved in interface execution, they support traceability, reasoning, and design documentation.

Overall, the Abstract User Interface metamodel provides a highly structured, extensible, and modality-independent framework for representing user interaction logic. It separates interaction semantics from implementation details while integrating data management, event handling, behavioral modeling, localization, security, and state management. This abstraction facilitates transformation into multiple concrete user interfaces and supports adaptive, context-aware, and reusable interface design within the UsiXML framework.

A.5 Concrete User Interface Metamodel

The Concrete User Interface (CUI) metamodel in UsiXML represents the user interface at the platform-specific level of abstraction. It corresponds to the Platform-Specific Model (PSM) in Model-Driven Engineering (MDE) and describes user interfaces in terms of concrete interaction units that are adapted to a particular interaction modality, platform, or device. Unlike the Abstract User Interface (AUI), which remains independent from implementation details, the CUI metamodel introduces modality-dependent characteristics such as graphical layout, visibility, styles, and interaction behaviors while still remaining independent from a specific programming language or toolkit.

The main objective of the CUI metamodel is to bridge the gap between abstract interaction concepts and executable user interfaces. It specifies how interface elements should appear and behave for a particular modality, such as graphical, vocal, tactile, or multimodal interaction. In the current version of the metamodel, the graphical modality is the most extensively defined.

At the core of the metamodel is the ConcreteUIModel, which represents the entire concrete user interface before transformation into executable code. It organizes the interface into ConcreteInteractionUnits that correspond to concrete interaction components adapted to a specific platform or modality. Although the model is platform-specific, it still remains independent of implementation technologies such as Java Swing, HTML, or VoiceXML.

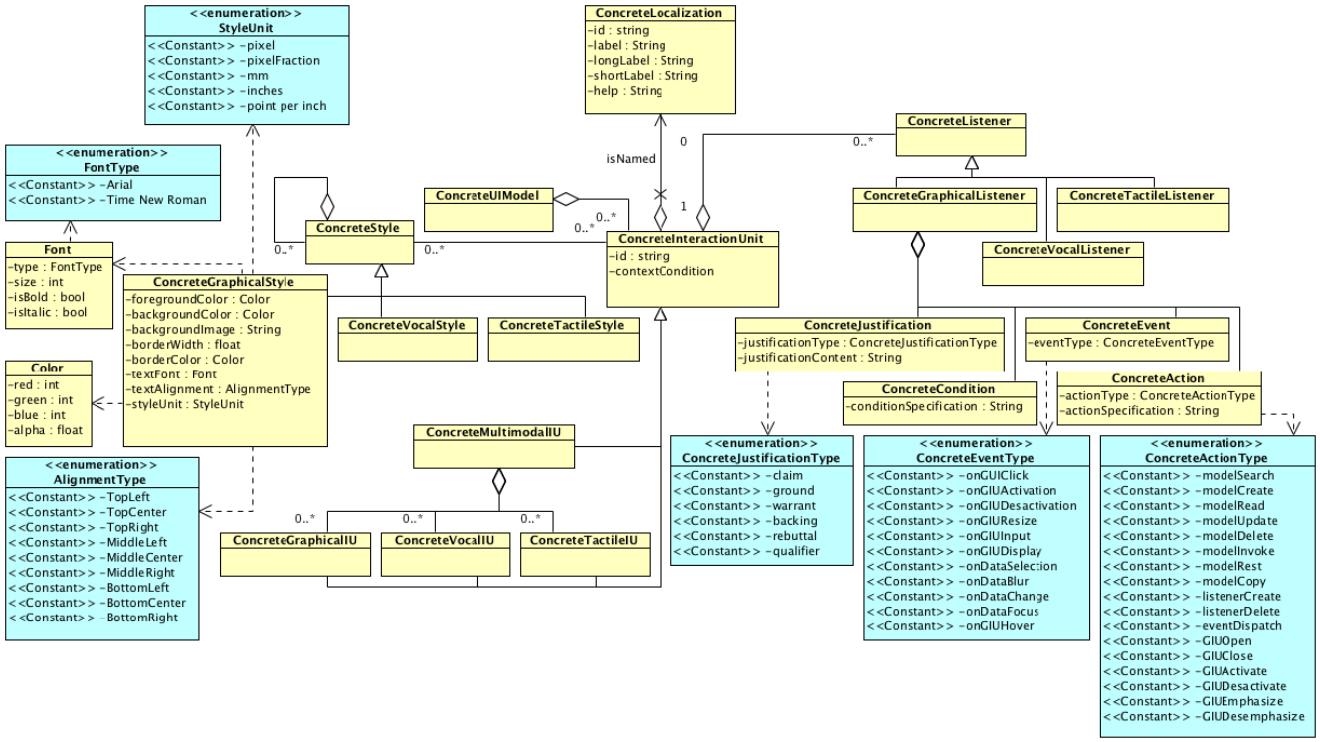


Fig. 29: Concrete User Interface metamodel as a UML class diagram (extracted from [1])

The principal entity of the metamodel is the ConcreteInteractionUnit. Every concrete component in the interface is represented by this entity. Concrete interaction units may belong to different modalities including graphical, vocal, tactile, or multimodal interfaces. Each interaction unit contains identifiers and contextual conditions that establish links with the Context model. This integration enables the interface to adapt dynamically to context changes such as device characteristics, user profiles, or environmental conditions.

To support multilingual interfaces, the metamodel defines ConcreteLocalization entities. These entities provide localized textual resources associated with concrete interaction units. Localization attributes include labels, long labels, short labels, and help descriptions. This mechanism ensures that the same interface structure can be rendered in different languages while preserving usability and consistency across cultural contexts.

A major specialization within the metamodel is the ConcreteGraphicalIU hierarchy, which models graphical user interface components. ConcreteGraphicalIU is the main graphical entity and represents visual interaction elements. Each graphical interaction unit includes spatial and presentation attributes such as position coordinates, dimensions, visibility, enablement, resizability, and splittability. These attributes allow the specification of concrete layout and rendering properties necessary for graphical interface design.

The graphical interaction units are divided into two major categories: ConcreteGraphicalCompoundIU and ConcreteGraphicalElementaryIU. Compound graphical interaction units act as containers that organize and group other graphical components. They support hierarchical organization and layout structuring of the interface. Elementary graphical interaction units represent atomic widgets or controls that users interact with directly.

The metamodel includes several predefined graphical widgets and containers that abstract common user interface components. Examples include tab bars, tabs, menus, dialog windows, panels, lists, forms, labels, buttons, and selection controls. These components are described independently from any specific toolkit while still capturing the characteristics required for graphical rendering and interaction.

Relationships among graphical interaction units are represented through ConcreteGraphicalRelationship entities. These relationships specify how graphical components are spatially organized and connected within the interface hierarchy. They define containment, alignment, and structural composition among widgets and containers.

The CUI metamodel also incorporates styling capabilities through ConcreteStyle entities. Concrete styles define visual appearance properties for interaction units such as colors, fonts, borders, spacing, alignment, and rendering preferences. By separating style information from structural definitions, the metamodel supports interface customization, theming, and adaptation to different devices or branding requirements.

Behavioral aspects of the concrete interface are managed through ConcreteListener entities. Similar to the listeners defined in the Abstract User Interface metamodel, concrete listeners implement Event-Condition-Action (ECA) rules that define how concrete widgets react to user interactions or system events. These listeners connect low-level interface events with actions affecting the interface state or the underlying domain model.

The metamodel also supports modality specialization. Although the graphical modality is the most detailed in the specification, the architecture allows extension toward vocal, tactile, and multimodal interaction models. This extensibility enables the integration of emerging interaction technologies while preserving the consistency of the overall UsiXML framework.

An important characteristic of the Concrete User Interface metamodel is its role in model transformations. The CUI serves as the intermediate representation between abstract interaction models and executable user interfaces. It refines abstract interaction

concepts into modality-specific components while maintaining traceability with task models, domain models, and context models. This separation of concerns improves maintainability, adaptability, and reusability in user interface engineering.

Overall, the Concrete User Interface metamodel provides a structured and extensible framework for specifying concrete interaction components, layouts, visual styles, localization resources, and behavioral mechanisms. By refining abstract interaction concepts into modality-aware representations, it enables the systematic generation of executable user interfaces adapted to multiple platforms, devices, and contexts of use within the UsiXML model-driven architecture.

A.6 Transformation Metamodel

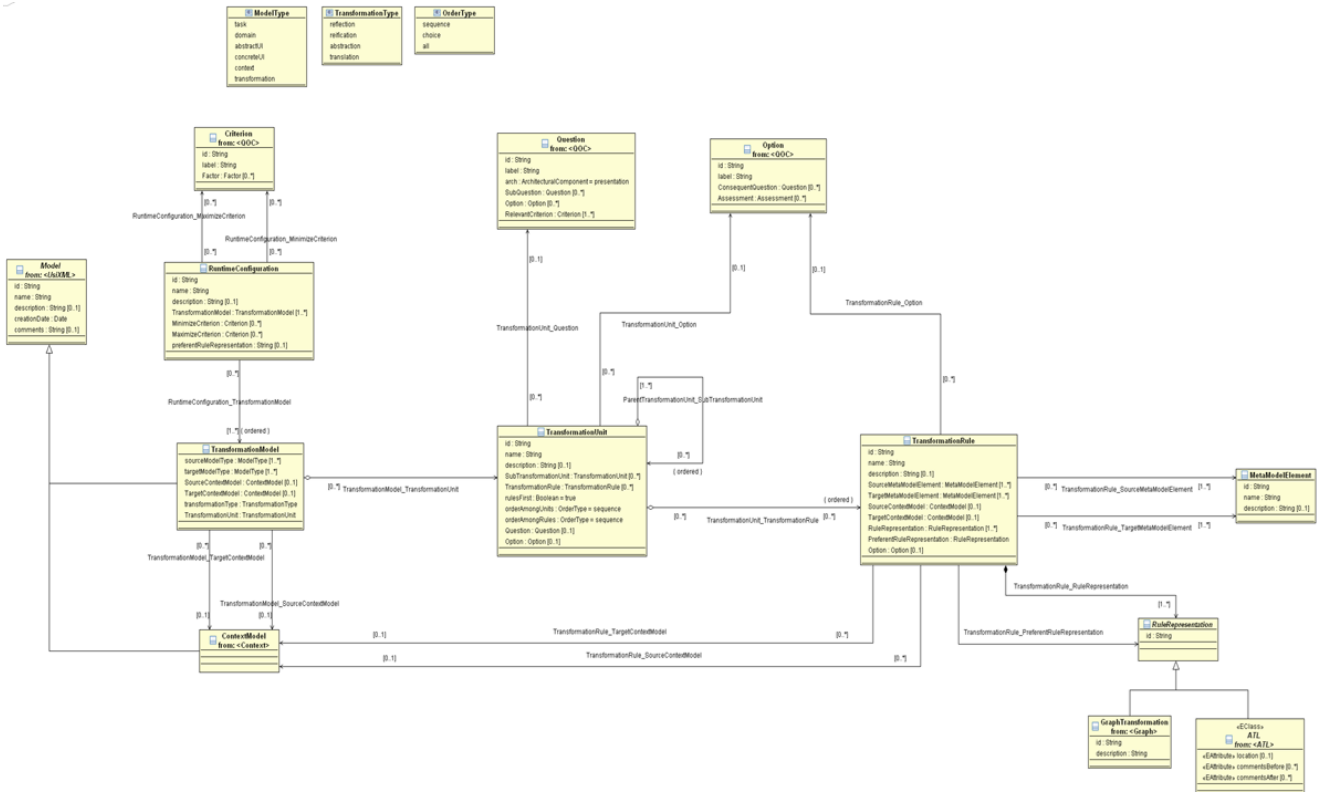


Fig. 30: Transformation metamodel as a UML class diagram

The transformation metamodel in UsiXML defines the mechanisms used to transform models from one level of abstraction to another within the model-driven engineering process. It plays a central role in connecting the different UsiXML metamodels, such as Task, Context, Domain, Abstract User Interface (AUI), and Concrete User Interface (CUI), enabling the systematic generation and adaptation of user interfaces across multiple platforms and contexts of use.

The transformation metamodel supports model-to-model transformations by specifying how elements from a source model are mapped, refined, or translated into elements of a target model. These transformations may occur vertically between abstraction layers, such as from task models to abstract user interfaces or from abstract interfaces to concrete interfaces, as well as horizontally between models at the same abstraction level but adapted to different contexts or platforms.

The metamodel is structured around transformation rules that define the conditions and operations necessary to produce target model elements. Each transformation rule specifies source elements, target elements, constraints, and transformation logic. The rules may include mappings between concepts, relationships, attributes, and behavioral properties across metamodels. This approach ensures consistency and traceability during the interface development process.

An important feature of the transformation metamodel is its support for context-aware adaptation. By integrating with the Context metamodel, transformations can dynamically generate or adapt user interfaces according to contextual factors such as user profiles, devices, environments, or interaction modalities. This capability enables adaptive and plastic user interfaces that can evolve according to changing conditions of use.

The transformation metamodel also incorporates concepts from Quality of Service and Quality of Use through dedicated packages. These packages allow transformations to take into account usability, performance, accessibility, and other quality criteria during the generation process. As a result, the produced interfaces are not only functionally correct but also optimized for user experience and operational constraints.

The metamodel is organized into several packages that separate concerns and improve extensibility. These include packages for context handling, transformation specification, and quality management. This modular structure allows developers to extend the

transformation framework with additional rules, operators, or adaptation strategies while maintaining consistency across the UsiXML ecosystem.

Another important aspect of the transformation metamodel is its compatibility with existing model transformation technologies such as ATL (Atlas Transformation Language) and graph transformation approaches. This compatibility facilitates interoperability with external model-driven engineering tools and supports automated transformation execution.

The transformation process itself may involve several operations, including abstraction, refinement, translation, composition, and synchronization of models. Transformations preserve relationships between source and target models, enabling traceability and reverse engineering capabilities. This traceability is essential for maintaining coherence across different abstraction levels and for supporting iterative interface development.

Overall, the transformation metamodel provides the formal infrastructure required to automate the evolution of user interface models throughout the UsiXML development lifecycle. By enabling structured, reusable, and context-sensitive transformations between models, it supports the creation of adaptive, consistent, and platform-independent user interfaces within a model-driven engineering framework.

A.7 Mapping Metamodel

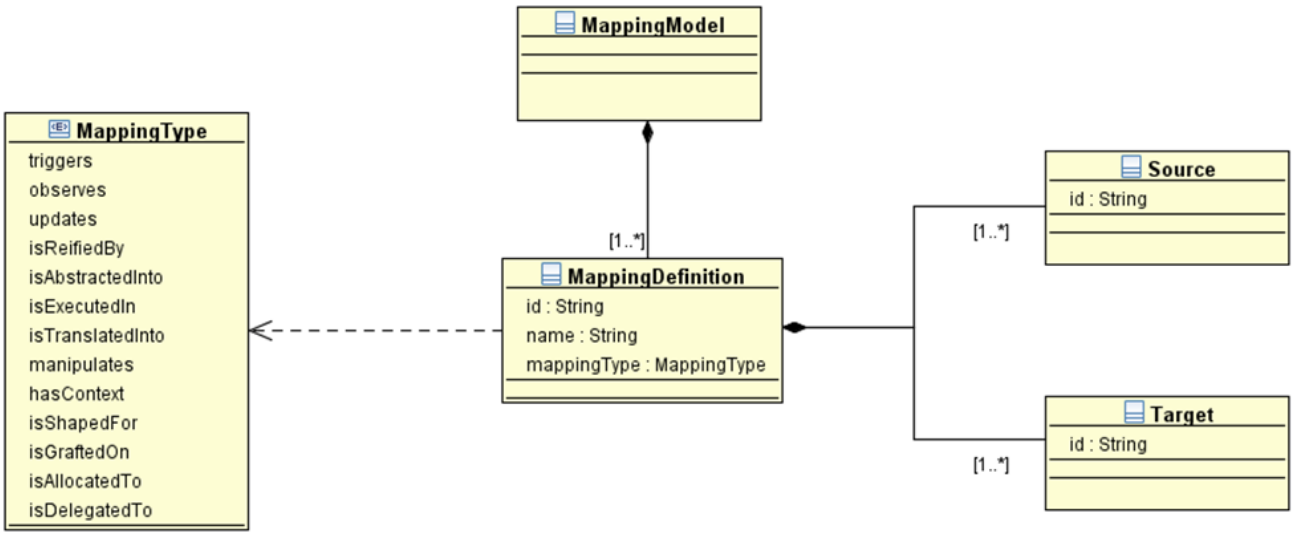


Fig. 31: Mapping metamodel as a UML class diagram (extracted from [1])

The mapping metamodel in UsiXML defines the relationships between elements belonging to different metamodels and abstraction layers. Its main purpose is to establish traceability and correspondence among models such as Task, Context, Domain, Abstract User Interface (AUI), and Concrete User Interface (CUI). Through these mappings, the framework can connect user tasks, domain concepts, contextual situations, and interface components in a coherent manner.

The mapping metamodel supports model-driven transformations by specifying how elements from one model are associated with elements in another model. For example, a task can be linked to an abstract interaction unit, or a contextual situation can be associated with a specific interface adaptation. This enables synchronization and consistency across the entire development lifecycle.

Overall, the mapping metamodel acts as an integration mechanism within the UsiXML framework, ensuring coordination, traceability, and interoperability between the various models involved in user interface design and adaptation.

A.8 Resource Metamodel

The resource metamodel in UsiXML describes the physical and logical resources involved in user interaction within an interactive system. It defines the characteristics and capabilities of devices, platforms, and interaction resources that support the execution of user interfaces. These resources may include hardware components such as screens, keyboards, microphones, speakers, sensors, and network connections, as well as software resources and interaction modalities.

The metamodel enables the specification of resource properties such as availability, capacity, performance, compatibility, and supported interaction modalities. By modeling these characteristics, the framework can adapt user interfaces according to the capabilities and limitations of the execution environment.

The resource metamodel is closely connected to the Context metamodel and supports context-aware adaptation and plasticity. It allows transformations and mappings to generate user interfaces optimized for different devices, platforms, and interaction conditions.