Data Science and Engineering (2020) 5:240–260
https://doi.org/10.1007/s41019-020-00131-3

Integrating Web-Based Collaborative Live Editing and Wireframing into a Model-Driven Web Engineering Process

Peter de Lange[1] · Petru Nicolaescu[1] · Alexander Tobias Neumann[1] · Ralf Klamma[1]

Received: 11 March 2020 / Revised: 30 April 2020 / Accepted: 8 June 2020 / Published online: 24 June 2020

Abstract
Today's Model-Driven Web Engineering (MDWE) approaches automatically generate Web applications from conceptual, domain-specific models. This enhances productivity by simplifying the design process through a higher degree of abstraction. Due to this raised level of abstraction, the collaboration on conceptual models also opens up new use cases, such as the tighter involvement of non-technical stakeholders into Web development. However, especially in the early design stages of Web applications, common practices for requirement elicitation mostly rely on wireframes instead of MDWE, created usually in analog settings. Additionally, state-of-the-art MDWE should integrate established and emerging Web development features, such as Near Real-Time (NRT) collaborative modeling and shared editing on the generated code. The combination of collaborative modeling, coding and wireframing, all in NRT, bears a lot of potential for improving MDWE practices. The challenge when covering these requirements lies with synchronizing source code, wireframes and models, an essential need to cope with regular changes in the software architecture to provide the flexibility needed for agile MDWE. In this contribution, we present a MDWE approach with live code editing and wireframing capabilities. We present the conceptual considerations of our approach, the realization of it and the integration into an overarching development methodology. Following a design science approach, we present the cyclic iterations of developing and evaluating our artifacts, which show promising results for collaborative Web development tasks that could open the gate towards novel, collaborative and agile MDWE techniques.

Keywords Collaborative Model-Driven Web Engineering · Collaborative live editing · Collaborative wireframing · Model-to-model synchronization · Model transformations

1 Introduction

Current *Model-Driven Web Engineering* (MDWE) approaches try to increase productivity by enabling the generation of Web applications, based on information usually specified in the form of conceptual models [21]. Corresponding to a certain domain-specific metamodel, the models reflect the structure of Web frontends and abstract the pagination and the navigation of applications. Based on certain templates and incorporated, framework-specific best practices, the resulting applications can be specified and instantiated accordingly. By splitting the metamodel into separate views that reflect separate parts of the application, different stakeholders can focus on different parts of application design, according to their background, expertise and interest. If used in a *Near Real-Time* (NRT) collaborative fashion, this approach bears the potential to involve non-technical stakeholders better into the development process and thereby also serves as a means to improve requirements elicitation.

However, modeling alone often cannot depict the complexity of a Web application. Certain parts of an application are very specific, and while a metamodel can enforce the overall architecture of a Web application, often manual code editing is still needed to implement the complete application functionality. To adapt to this, a collaborative MDWE approach has to support development cycles with

✉ Peter de Lange
lange@dbis.rwth-aachen.de

Petru Nicolaescu
nicolaescu@dbis.rwth-aachen.de

Alexander Tobias Neumann
neumann@dbis.rwth-aachen.de

Ralf Klamma
klamma@dbis.rwth-aachen.de

[1] Chair for Information Systems and Databases, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany

rapid changes in the model-based architecture and the corresponding source code, both being simultaneously edited. Hence, traditional methods that enable the synchronization between model and code need to be adapted to this collaborative setting.

On the other hand, modeling (and especially manual code editing) still requires a rather good and specific development knowledge, in order to be able to model and modify the generated software artifacts. Software prototyping, often also called wireframing, is a popular software engineering method to quickly conceive the most important aspects of a software application at the early stages of software development. It is a collaborative and social process, that involves designers, end users, developers and other stakeholders. In contrast to a conceptual model, that consists of rather abstract nodes and edges, a wireframe provides a closer representation of the final Web application's visual design. Consequently, a wireframe is more intuitive and feels more familiar to non-technical stakeholders. Such an application promises a lower learning curve, with less required knowledge about Web development. In order to achieve such a novel collaborative frontend development practice, live synchronization between models and wireframes have to be implemented.

To illustrate this concept, we want to sketch a use case that integrates this novel MDWE practice. A professional community of medical doctors uses videos and images as main study and documentation objects in their training practice. We now assume that this community wants to integrate 3D objects (e.g., highly detailed digital representations of anatomical objects) in their training practices. Such features cannot be easily implemented without technical knowledge. On the other hand, they are also hard to explain to developers without deeper domain knowledge. Thus, the community uses a Web-based MDWE approach for requirements-elicitation with (possibly external) developers. Doctors and developers can now distribute according to their domain-specific knowledge to work on the corresponding views. For example, doctors could produce wireframes to explain the developers their proposed extension of the current system. Directly transforming these wireframes into models, developers start working on the corresponding models and source code, all directly in the browser and in NRT. At all times, the Web application is automatically generated and deployed on the Web, thus the community can follow along and provide direct feedback on the current state of the prototype.

In this contribution, we present a Web-based MDWE approach that integrates both live code editing and wireframing, all in a NRT collaborative setting. This work provides a first complete view on the approach, including the interplay between the different, previously independently published parts [8–12]. We present additional evaluations, the embedding into an overarching development methodology, and a description of the underlying research methodology and its application in this research project. We start by presenting the background and related work of our contribution in Sect. 2. Our research is based on a design science methodology [17], that is presented in Sect. 3. We present a conceptual overview of our approach in Sect. 4, which includes the presentation of the different views and representations of a Web application within our framework (Sect. 4.1), as well as the general application metamodel (Sect. 4.2). The approach is embedded into an overarching development methodology for the creation and deployment of *peer-to-peer* (p2p)-based microservices. We describe this integration in Sect. 4.3, which also includes the connection to a Web-based requirements analysis platform and the possibility to directly define monitoring capabilities of services within the MDWE environment. We apply related work on traceability [29] and synchronization [16] to realize the live code editing functionality (Sect. 5.1). We adapt related conceptual mappings of MDWE [23] and wireframes [35], and present a conceptual mapping for the co-evolution of models and wireframes in NRT (Sect. 5.2). Our approach is realized as a MDWE framework named *Community Application Editor* (CAE), which is described in Sect. 6. Here, we describe the user interface of the CAE (Sect. 6.1), its general architecture (Sect. 6.2) and the technical integration of both live code editing (Sect. 6.3) and wireframing (Sect. 6.4). We continue by giving a detailed overview on the evaluations we conducted during our research in Sect. 7, before we conclude this contribution in Sect. 8.

2 Background and Related Work

In this section, we describe the work related to our research. We start with an overview on MDWE in Sect. 2.1, before we present the background on transformation algorithms, that we use for the realization of the live coding features (Sect. 2.2). We conclude this section with related work on wireframing, which includes works that define a structural user interface model, that we took as a basis to realize the collaborative wireframing features (Sect. 2.3).

2.1 Model-Driven (Web) Engineering

Most MDWE approaches follow the philosophy of separation of concerns [18]. Based on a comprehensive metamodel, certain views are defined to reflect specific aspects of a Web application. One of the first MDWE approaches that obeyed the separation of concerns idea in MDWE was OOHDM [36], with the goal of dealing with the increasing complexity of Web applications. It described a methodology for systematic guidance to design large scale, dynamic Web applications. The main activities of the OOHDM

methodology comprised a conceptual, navigational and abstract user interface design and proposed how they are implemented in the final Web application. A slightly more recent, as well as ongoing, MDWE methodology is the *UML-based Web Engineering* (UWE) [20], which was conceived as a conservative extension of the UML. Thus, already existing concepts of the UML are not modified, the new extensions are just related to existing concepts. The first extensions are UML stereotypes, which are used to define new semantics for model elements, e.g., a navigation link. The *Object Constraint Language* (OCL) is used to define constraints and invariants for classes. UWE follows the separations of concerns principle to split up the modeling process into the conceptual, navigational, and presentation modeling part. WebML is another MDWE approach developed in 2000. It does not propose another language for data modeling, but also extends the UML and is compatible with classical notations of ER-diagrams and others [5]. WebML as well emphasizes the concept of separation of concerns. Therefore, the development process is divided into four distinct modeling phases. The structural model represents the content of the site expressed as UML class or ER diagram, and the hypertext model consists of a composition and navigation model. The former one describes which entities of the structural model are composed by a certain page and the latter one specifies the links between pages. The third one is the presentation model which expresses the layout and graphical appearance of pages. Finally, the personalization model defines user and/or user group specific content. In 2013, WebML emerged into the *Interaction Flow Modeling Language* (IFML) [4] and was adopted as a standard by the *Object Management Group* (OMG). A rather recent implementation of the IFML specification in a Web-based editor is the Direwolf Model Academy [22], which also features NRT collaborative editing of IFML models.

ArchiMate is an enterprise architecture modeling language [23]. Although not a direct MDWE approach, it is relevant related work for our approach, because of its interpretation of the separation of concerns paradigm. It separates the content and visualization of the view. The main advantage of this is the usage of different visualizations on the same modeling approach and vice versa. The content of a view is derived from the base model and expressed in the same modeling concept. The visualization on the other hand can be completely different from the actual representation of the model. ArchiMate allows to define a set of modeling actions, that alter the content of the model. These modeling actions are mapped to operations on a specific visualization of the view. This additional abstraction level allows to define any sort of visualization, like videos or dynamic charts.

In this contribution, we use the concept of view separation to map certain operations on the wireframing editor to operations on the modeling canvas, which alter the current state of the wireframing, respectively, the modeling view. To our knowledge, there exists no implementation of a MDWE framework that allow for a complete cycle of collaborative modeling, coding and deployment of an application on the Web in NRT.

2.2 Transformation Algorithms

In the scope of MDWE, *Model to Text* (M2T) transformations are a special form of *Model to Model* (M2M) approaches, in which the target model consists of textual artifacts [25], in this case the source code of the generated Web application. The target model is generated based on transformation rules, defined with respect to a model's metamodel [24]. Template-based approaches are (together with visitor-based approaches) the most prominent solution for M2T transformations [6]. Here, text fragments consisting of static and dynamic sections are used for code generation. While dynamic sections are replaced by code depending on the parsed model, static sections represent code fragments not being altered by the content of the parsed model [28]. An important aspect of M2T transformations is model synchronization. It deals with the problem that upon regeneration, changes to the source model have to be integrated into the already generated (and possibly manually modified) source code. To achieve this, traces are used to identify manual source code changes during a M2T (re)transformation. In MDWE, managing traceability has evolved to one of the key challenges [2]. Another challenge is the decision on the appropriate granularity of traces, as the more detailed the links are, the more error-prone they become [15, 37]. Formal definitions of model synchronization for M2M transformations have been proposed in [14, 16].

2.3 Wireframing

In Web engineering, a wireframe is an agile prototyping technique to sketch the skeletal structure of a Web application [3]. There exist a plethora of wireframing and mockup tools on the Web. We here exemplary introduce Balsamiq[1] (as one of the most used ones) and Mockingbird[2] (as it features NRT collaboration and is Web-based). The idea behind Balsamiq is not to build large and fully interactive prototypes, which take hundreds of hours to develop and may lead to costly refinements if something can not be realized as intended. Instead, Balsamiq follows a more rapid development philosophy. This has the advantage that developers gain experience and evaluate components of the wireframe directly on a very early version of the Web application,

[1] https://balsamiq.com.

[2] https://www.gomockingbird.com.

which can also involve end user feedback. This feedback is used to tweak the wireframes and the implementation process starts again. Therefore, Balsamiq offers only limited interactivity features on a wireframe. Mockingbird is a Web-based wireframing application that offers NRT collaborative editing. The graphical editor offers the most common UI elements of today's Web applications, which can be rearranged and resized freely on a page. Similar to Balsamiq, it is possible to link pages and preview them to demonstrate the Web application's interactivity flow.

Mockup Driven Development (MockupDD) is a hybrid, model-based and agile Web engineering approach [35]. The main goal of MockupDD is to extract and combine the advantages of MDWE methodologies and the rapid collaborative design process of wireframing, to add agility to existing MDWE approaches. MockupDD describes a transformation approach from a mockup to a comprehensive model that is further transformed to the specific models of an arbitrary MDWE approach. In most related approaches, wireframes are not considered as models, and their impact declines in later development stages. MockupDD tackles this with a generic approach to integrate mockups directly into the whole MDWE development process. An additional computational instance builds the bridge from the output of an arbitrary wireframing tool to an arbitrary MDWE approach. The MockupDD methodology begins by creating UI mockups with an arbitrary tool, e.g., Balsamiq or Mockingbird. The resulting mockup file is then parsed, validated and analyzed with regard to a *Structural UI* (SUI) metamodel, which denotes each UI control element, their compositions and hierarchical structures. The goal is to obtain a "sufficient enough" structural model of the UI. Based on this SUI model, another transformation approach to the specific model of the used MDWE methodology is required. To further enrich the representational strength of a SUI model, MockupDD includes a tagging mechanism. A tag is simple specification that is applied over a concrete node of the SUI model and consists of a name and an arbitrary number of attributes. The main purpose of a tag is to define functional or behavioral aspects of a certain UI element. It allows the designer to construct more complex wireframe specifications. An UI element may have an arbitrary number of tags assigned to it. A SUI model enriched with tags is also called a *SUIT* model. The concept of MockupDD has been adapted to various modeling languages and domains (WebML [35], UWE [33], IFML [34], and specifically focusing on mockups of touch user interfaces [1]). We base our wireframing integration partly on the conceptual findings of MockupDD, and take the approach one step further by co-evolving the wireframe and MDWE artifacts throughout the whole Web application development process. To our knowledge, there exists no approach that both enables Web-based wireframing, while simultaneously providing a way to transfer the wireframes to conceptual models (or code) and vice versa. As such, our approach combines the conceptual findings of research in the domain of wireframe-to-model transformations with the applicability of existing Web-based wireframing solutions.

3 Methodology

Our methodology follows a design science approach as proposed by Hevner [17], and applies the guidelines proposed by Peffers [30]. Figure 1 depicts this process, which consists of seven iterations. We started with the initial question, how to integrate end users more into development, to close the gap in requirement elicitation. This led to the development of the initial CAE prototype, which we used to redesign an existing Web application that showcased its usability. These results were communicated in a demo paper [8]. The first usage of the CAE clearly pointed out that a more defined development process was needed. Thus, we started to create the agile and cyclic development process that the CAE approach currently implements. We first evaluated this approach in multiple evaluation sessions with teams of mixed professions, as well as that we observed the usage of it within a longer timespan in a university software development lab course. Results were communicated in [9, 10]. These evaluations showed a lack of expressiveness of the modeling language for certain aspects of a Web application, which we tackled by developing the Live Code Editor. The results of the evaluation with student developers were published in [12]. As both the Live Code Editor and the collaborative modeling are still rather abstract, especially in frontend development, our next step was the integration of the collaborative Wireframing Editor, which we published in [11]. We continued measuring the impact of the Wireframing Editor by evaluating the time spent in different views of the CAE, and finally we embedded the CAE even further into its overarching methodology by implementing service success measurement support. The combined results are published in this contribution.

4 Conceptual Overview

In this section, we provide an overview on our approach as a whole (Sect. 4.1), its underlying metamodel (Sect. 4.2), and its integration into an overarching development methodology (Sect. 4.3).

4.1 View-Based Model-Driven Web Engineering

Our approach follows the separation of concerns principle [18] and defines four orthogonal views [26] for the modeling of Web applications, based on a comprehensive metamodel:

– The *frontend component view*, represented by a model of a Web component

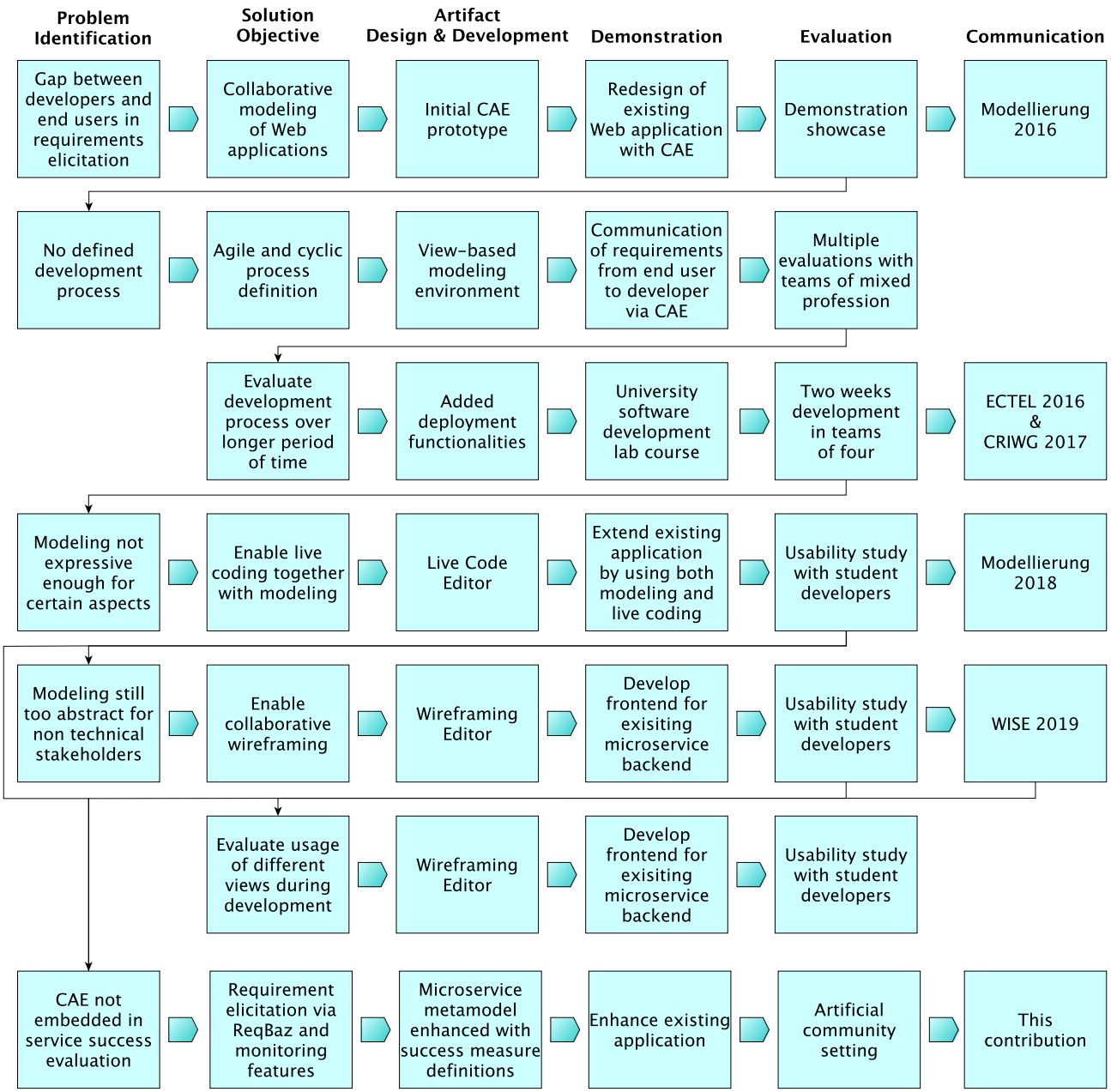


Fig. 1 Design science methodology we followed in conducting this research

- The *backend component view*, represented by a model of a microservice
- The *wireframing view*, a visual representation of a frontend component
- The *application view*, the overarching metamodel of the complete Web application

Figure 2 gives an overview of the different views, as well as their connections with each other and the code refinements/deployment. It can be split up into three main phases, namely the *Modeling*, the *Coding*, and the *Wireframing* phase. Based on this modeling—coding—wireframing cycle, the approach enables the collaborative, model-driven creation of Web applications.

To illustrate this concept, Fig. 3 depicts four representations of the same frontend component. Figure 3a shows the conceptual model. Figure 3b depicts the wireframe visualization of the frontend component model. Both the wireframe model and the conceptual model are used as input for the code generation to generate the code artifacts depicted in the Live Code Editor of Fig. 3c. Finally, Fig. 3d shows a live

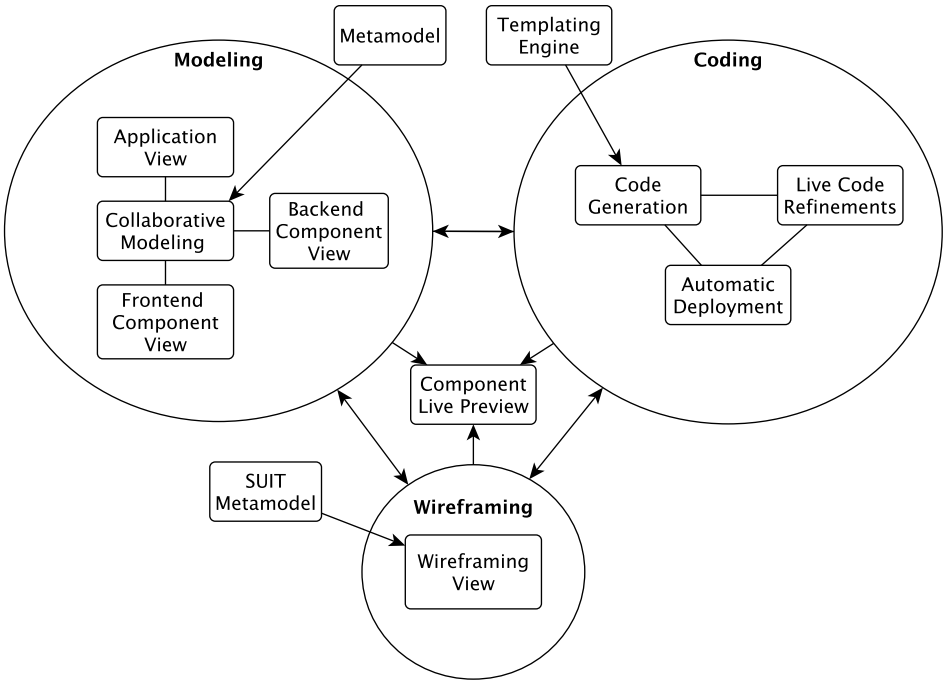
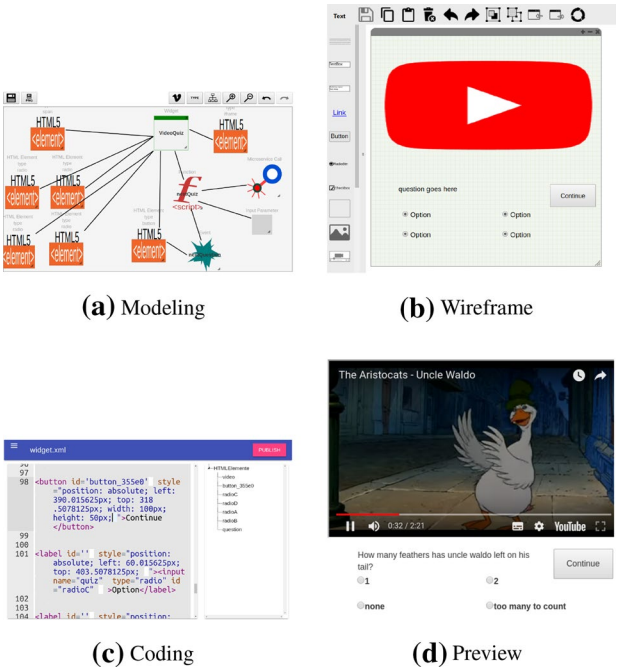


Fig. 2 Overview of the MDWE approach



(a) Modeling **(b)** Wireframe **(c)** Coding **(d)** Preview

Fig. 3 Different representations of the same frontend component

preview of the resulting application, based on the generated code artifacts.

4.2 A Web Application Metamodel

Although our general approach could be used for arbitrary MDWE frameworks and Web applications, in the scope of these works we consider Web applications composed of HTML5 and JavaScript frontends, and RESTful microservice backends. Figure 4 depicts this Web application metamodel. The central entity of a microservice is a *RESTful Resource*. It contains *HTTP Methods*, which form the interface for communication either via a RESTful approach, but also via an *Internal Service Call* from one HTTP method to another, possibly between different microservices. To enable service monitoring, each HTTP method can be enhanced with multiple *Monitoring Messages*. According to the idea of polyglot persistence, each microservice can have access to its own *Database* instance.

The central entity of a frontend component is a *Widget*. This widget consists of *Functions* and *HTML Elements*. HTML elements can either be *static*, meaning that they are not modified by any other element or functionality of the component, or *dynamic*, meaning that they either are created or updated by one of the frontend component's elements. Both static and dynamic HTML elements can trigger events, which can for example be a mouse click, that cause function calls. The second option to trigger a function call is via an *Inter Widget Communication (IWC) Response* object, that waits for an *IWC Call* to be triggered. These calls are again part of a function, which initiates them. A function is able to update or create a dynamic HTML element. The last part of the frontend component view is the communication and collaboration functionality, which includes the already mentioned IWC call response mechanism, as well as microservice calls that are triggered by a function. HTML elements can also be instrumentalized with collaborative support, making it possible for elements to share the same

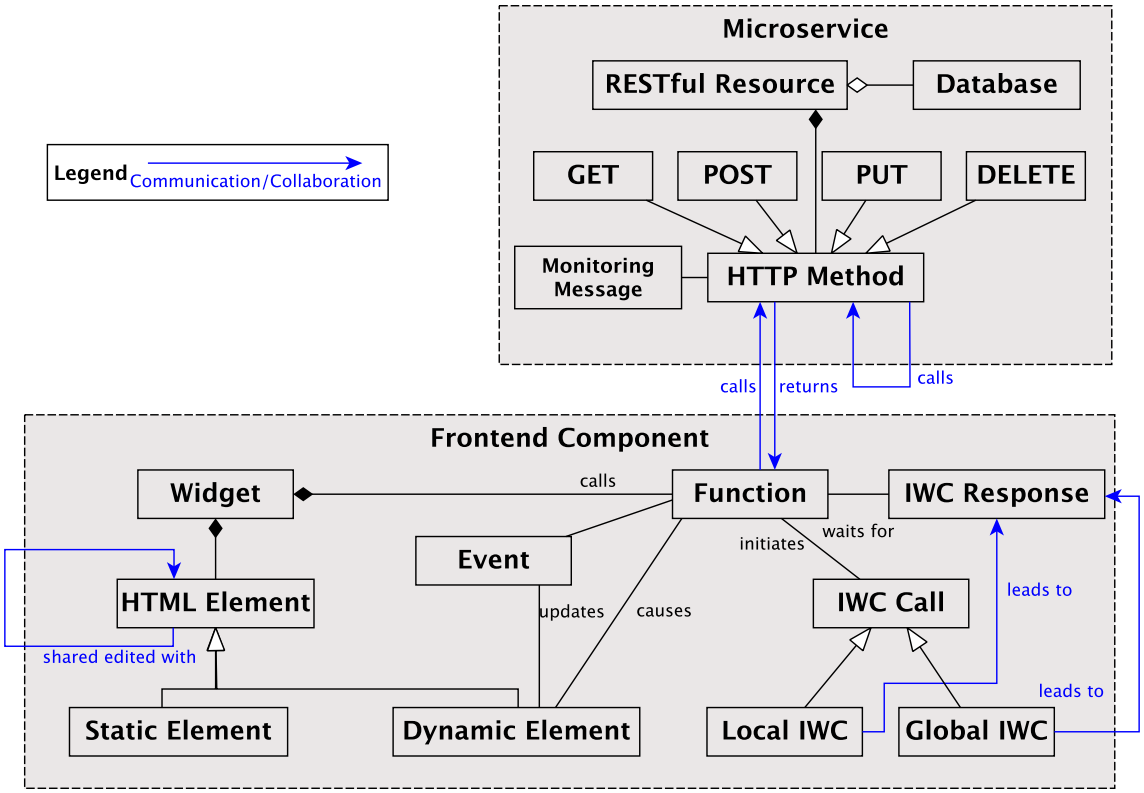


Fig. 4 The Web application metamodel used in this contribution

state/content in the Web browser of all participating users, propagating changes in NRT.

4.3 Embedding in Open Source P2P-Based Community Service Development Methodology

We integrated our MDWE framework into an agile methodology for distributing community microservices in an open source, decentralized p2p infrastructure [7]. The central part of the methodology is the p2p microservice framework itself, called las2peer [19], which consists of microservice hosting nodes. Additionally, a monitoring and evaluation suite [32] collects monitoring information in the network in a decentralized manner and can provide it centrally at a specified monitoring node. Our MDWE framework integrates itself as a means for speeding up development and enforcing best practices, with the possibility for continuous deployment of the developed services directly in the infrastructure.

To connect the framework more tightly with the overarching las2peer methodology and to further support requirement elicitation, we recently integrated the Requirements Bazaar [31], a Web application that is especially targeted at including end users in requirement gathering and discussion. A modeling space can be connected with a corresponding project in the Requirements Bazaar, and requirements can directly be discussed during the modeling process in a special component of the editor. Additionally, we also created modeling elements for different types of *Monitoring Messages*. These can be used to measure certain success features of a service, according to las2peer's monitoring and evaluation suite. Specifically, *Monitoring Messages* can be connected to an *HTTP Method*, which then log its processing time, to an *HTTP Payload*, which logs the payload content, or to an *HTTP Response*, to log the response content. These measures are available to be extended in the Live Code Editor or to be directly used in the service success modeling [19].

5 Conceptual Integration of Live Code Editing and Wireframing Support

In this section, we describe the formal conceptual integration of both the live code editing (Sect. 5.1) and the wireframing support (Sect. 5.2).

5.1 Model Synchronization Strategies for Live Code Editing

We unify the architecture of applications developed with our approach through the usage of protected segments, that enforce a certain base architecture, facilitating future service

and frontend orchestration, maintenance and training efforts for new developers. Protected segments in the source code describe a functionality that is reflected by a modeling element. In order to encourage the reuse of software components, we allow changes which modify the architecture only in modeling phases. Since our approach offers a cyclic development process, this can be done instantly by switching to modeling, changing the corresponding element and returning to a new coding phase. To further enforce this methodology, before source code changes are persisted, a model violation detection is performed. This informs the user about source code violating its corresponding model, e.g., architecture elements manually added to the source code instead of being modeled. Concerning the synchronization between the code and the model, our collaborative MDWE process uses a trace-based approach. Changes in the code produce traces, which are used in the model-to-code (re)generation in order to keep the corresponding code synchronized to the model elements. This way, the process can be reflected without the need to implement a full RTE approach.

Our general synchronization concept (depicted in Fig. 5) is divided into two separate synchronizations: a synchronization between the source code and its trace model and a second synchronization between the source model and the source code. In the following, we explain our synchronization concept by using a simple formalization. We denote the source models by S_i, source code models by T_i and trace models by tr_i. The source- and source code-metamodels are denoted by M_S and M_T. We use the definition of synchronization expressed in [16] as follows: two models A and B with corresponding metamodels M_A and M_B are *synchronized*, if

$$trans(A) = strip(B, trans) \tag{1}$$

holds for the transformation $trans : M_A \to M_B$ and a function $strip : M \times (M_A \to M_B) \to M$ that reduces a model of M with either $M = M_A$ or $M = M_B$ to only its elements relevant for the transformation. This definition uses the trans and strip functions [16]. Intuitively, the *trans* function expresses that applying a transformation to the source model yields the target model. The function *strip* is used to remove any additional elements and map models to only the relevant source/target model.

Synchronization of Source Code and Trace Model Based on a first model S_1, an initial generation of the source code T_1 and its trace model tr_1 is performed. As depicted in Fig. 5, the trace model tr_i is updated, once the source code changes. ΔT_{2i-1} are applied to the source code T_{2i-1} in the ith code refinement phase. Formally, a single source code change can be denoted by one of the two functions $\delta^+_{M_T} : M_T \times C \times \mathbb{N} \to M_T$ and $\delta^-_{M_T} : M_T \times C \times \mathbb{N} \to M_T$. While the former inserts a character $c \in C$ at position $n \in \mathbb{N}$, the latter deletes a character c from position n in the source code. Then, the result of applying the source code changes ΔT_{2i-1} on T_{2i-1} is defined by

$$\begin{aligned} T_{2i-1}\Delta T_{2i-1} &:= \delta^{\pm}_{M_T}(\delta^{\pm}_{M_T}(\cdots \delta^{\pm}_{M_T}(T_{2i-1}, c_1, n_1) \\ &\cdots, c_{k-1}, n_{k-1}), c_k, n_k) =: T_{2i} \end{aligned} \tag{2}$$

for $n_k, \in \mathbb{N}$, $c_k \in C$ and $k \in \mathbb{N}$.

Considering Eq. 1, the condition $trans(T_{2i}) = strip(tr_i, trans)$ must hold for the synchronization between the updated source code T_{2i} and trace model tr_i:

$$trans(T_{2i}) = strip(tr_i, trans) \tag{3}$$

$$\iff trans(T_{2i-1}\Delta T_{2i-1}) = strip(tr_i, trans) \tag{4}$$

$$\begin{aligned} &\iff trans(\delta^{\pm}_{M_T}(\delta^{\pm}_{M_T}(\cdots \delta^{\pm}_{M_T}(T_{2i-1}, c_1, n_1) \\ &\cdots, c_{k-1}, n_{k-1}), c_k, n_k)) \\ &= strip(tr_i, trans) \end{aligned} \tag{5}$$

For the synchronization between source code and trace model, we only need to update the lengths of the segments of the trace model. Therefore, we assume $strip(tr_i, trans) = len(tr_i)$, where $len(tr_i)$ is a tuple containing the segments' lengths. This leads to the following equation that must hold after the source code was updated:

$$\begin{aligned} &trans(\delta^{\pm}_{M_T}(\delta^{\pm}_{M_T}(\cdots \delta^{\pm}_{M_T}(T_{2i-1}, c_1, n_1) \\ &\cdots, c_{k-1}, n_{k-1}), c_k, n_k)) = len(tr_i) \end{aligned} \tag{6}$$

To satisfy this condition, each source code change needs to update the length of the segment that is affected by the deletion or insertion. Therefore, each $\delta^{\pm}_{M_T}$ is transformed to an update of the trace model tr_i:

$$\begin{aligned} &\delta^{\pm}_{M_T}(\delta^{\pm}_{M_T}(\cdots \delta^{\pm}_{M_T}(T_{2i-1}, c_1, n_1) \cdots, c_{k-1}, n_{k-1}), c_k, n_k) \\ &\quad \to \delta^{\pm}_{len}(\delta^{\pm}_{len}(\cdots \delta^{\pm}_{len}(len(tr_i), n_1) \cdots, n_{k-1}), n_k) \end{aligned} \tag{7}$$

$$with \quad \delta^{+}_{len}((l_1, \cdots, l_m), n) := (l_1, \cdots, l_j + 1, \cdots, len_m) \tag{8}$$

$$\delta^{-}_{len}((l_1, \cdots, l_m), n) := (l_1, \cdots, l_j - 1, \cdots, len_m) \tag{9}$$

where $l_i \in \mathbb{N}$ for $i, m \in \mathbb{N}, 1 \leq i \leq m$ is the length of the ith segment and $l_j, j \in \mathbb{N}$ for $1 \leq j \leq m$ is the length of the segment that is affected by an insertion or deletion in the source code at position n.

Synchronization of Model and Source Code In the model synchronization process, the last synchronized model S_i, the updated model S_{i+1}, the current trace model and the last synchronized source code T_{2i} are involved. By using the trace model of S_i, the applied model changes ΔS_i can be merged into the last synchronized source code T_{2i} without overwriting already implemented code refinements. As a result of the

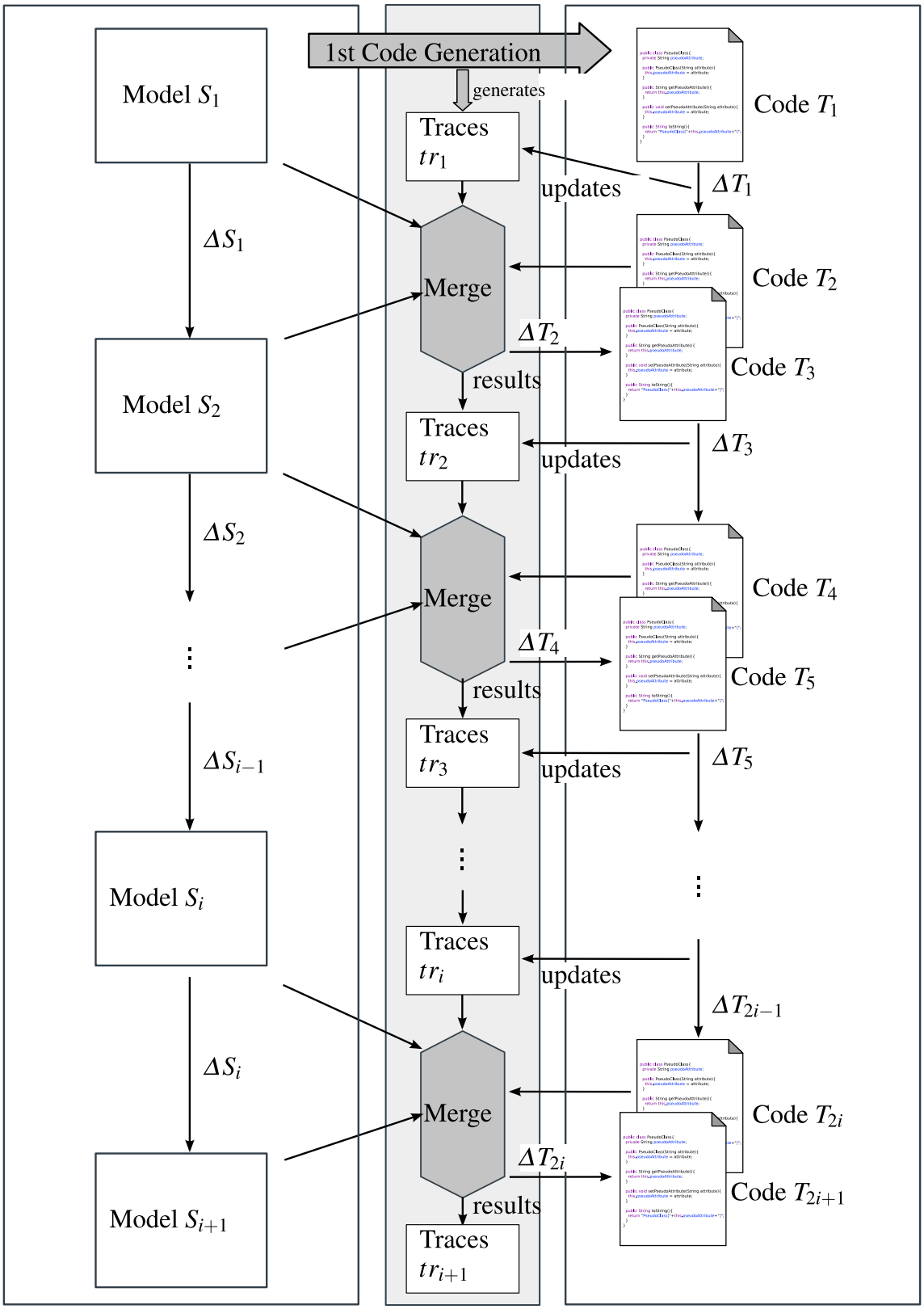


Fig. 5 Model synchronization

model synchronization, the updated source code T_{2i+1} and its trace model are obtained.

In general, model changes can be defined as functions of the form $\delta : M_S \rightarrow M_S$. More specifically, the model changes can be denoted by the following five functions, adapted from [16]: δ_t^+ , δ_t^- : creating/deleting element of type t; $\delta_{e,s1,s2}^+$, $\delta_{e,s1,s2}^-$: adding/deleting edge from element s1 to s2; and $\delta_{a,s1,v}^{attr}$: setting attribute a of element s1 to value v. As such, applying ΔS_i to S_i can be defined as a sequence of these changes:

$$S_i \Delta S_i \ := \delta_1 \circ \cdots \circ \delta_n(S_i) =: \ S_{i+1} \tag{10}$$

According to Eq. 1, the following equation must hold for the synchronization between model and source code:

$$trans(S_{i+1}) = strip(T_{2i+1}, trans) \tag{11}$$

$$\iff trans(S_i \Delta S_i) = strip(T_{2i+1}, trans) \tag{12}$$

$$\iff trans(\delta_1 \circ \cdots \circ \delta_n(S_i)) = strip(T_{2i+1}, trans) \tag{13}$$

Furthermore, as all parts of the source code that directly correspond to model elements are contained in protected segments, we assume $strip(T_{2i+1}, trans) = prot(T_{2i+1})$, where $prot(T_{2i+1})$ represents the source code that is reduced to the content of its protected segments. Finally, this leads to the following equation that must hold after the synchronization process:

$$trans(\delta_1 \circ \cdots \circ \delta_n(S_i)) = prot(T_{2i+1}) \tag{14}$$

To satisfy this equation, each individual model change $\delta_i, i \in \mathbb{N}, 1 \leq i \leq n$ is transformed to its corresponding source code changes. Next, we first introduce formulas that are needed for the later transformations.

Attribute value: the value of the attribute labeled *name* of a model element *elm* is denoted by $attr_{name}(elm) := (c_1, \cdots, c_k)$ with $c_i \in C$ for $i, k \in \mathbb{N}, 1 \leq i \leq k$.

Position and length of an element: the position of the first character of a model element *elm* within a file is defined by $pos_{seg}(elm)$. The length of *elm* is defined by $len_{seg}(elm)$.

Position and length of an attribute: the position of the first character of an attribute a of a model element *elm* is defined by $pos_{attr}(a, elm)$. The length of a is defined by $len_{attr}(a, elm)$.

Template: a template for an element *elm* of type t is denoted by

$$temp_t(attr_{name_1}(elm), \cdots, attr_{name_n}(elm)) := (c_1, \cdots, c_k)$$

with $c_i \in C$ for $k, i \in \mathbb{N}, 1 \leq i \leq k$. The attributes are used for the instantiation of the variables occurring in the template. We further define two functions that ease the formulas for deleting and inserting multiple characters:

$$\delta^{*+}(T, (c_1, \cdots, c_k), n) := \delta_{M_T}^+(\cdots \delta_{MT}^+(T, c_k, n+k) \cdots, c_1, n) \tag{15}$$

$$\delta^{*-}(T, n, k) := \delta_{M_T}^-(\cdots \delta_{M_T}^-(T, c_{n+k}, n+k) \cdots, c_n, n) \tag{16}$$

While the former inserts a tuple of characters starting from position n into a file, the later deletes the characters $c_n, ..., c_{n+k}$ at the positions $n, ..., n+k$ from a file. The transformation of model to source code changes is highly dependent on the type of the updated model elements. An elaborate example of such a transformation can be found in [12].

5.2 Mapping Between SUIT Wireframing- and Frontend Component Model

Inspired by the concepts of MockupDD and following the view separation of ArchiMate (both presented in Sect. 2.3), we developed a SUIT model for the wireframing integration and defined the transformations of this model to the frontend component metamodel, depicted in Fig. 6. The SUIT model of the wireframing editor comprises the most common HTML elements of the current HTML5 standard. It offers simple structural elements like buttons, text boxes and containers. Furthermore, media elements like the HTML5 video and audio player and custom Polymer elements[3] are supported. Also compositions of elements are defined, like a checkbox with a label. Each UI control element of the SUIT model has its own set of attributes defined, according to the HTML5 standard. We also introduced a so-called SharedTag, that can be assigned to any UI control element to add NRT collaborative behavior to it.

Conceptually, an instance of the SUIT model is a labeled tree. We formally define such a tree as a connected, acyclic and labeled graph. An arbitrary element $v \in V$ always has the signature $v = (l, t, A)$, where $l \in \Sigma$ is the label, with Σ being a finite alphabet of vertex and edge labels. $t \in T$ is the type of the node, where T is either a UI control element or a tag defined in the SUIT metamodel, as depicted in Fig. 6. For example T might consist of the following elements: $UI = \{Text, Button, Video, Canvas, ..\}$ and $Tag = \{SharedTag, DynamicTag, ..\}$ with $T = UI \cup Tag$. A is a finite set of properties related to an UI control element or tag and each $a \in A$ is a key-value-pair (k, v) with $k, v \in \Sigma$. The tree always consists of a distinguished vertex r, which is also called the root. The root is always of type *Widget*. The $parent(v)$ function is a helper function that yields the parent vertex for a vertex of the SUIT tree. If the vertex v is the root, the root will be returned.

[3] https://www.polymer-project.org.

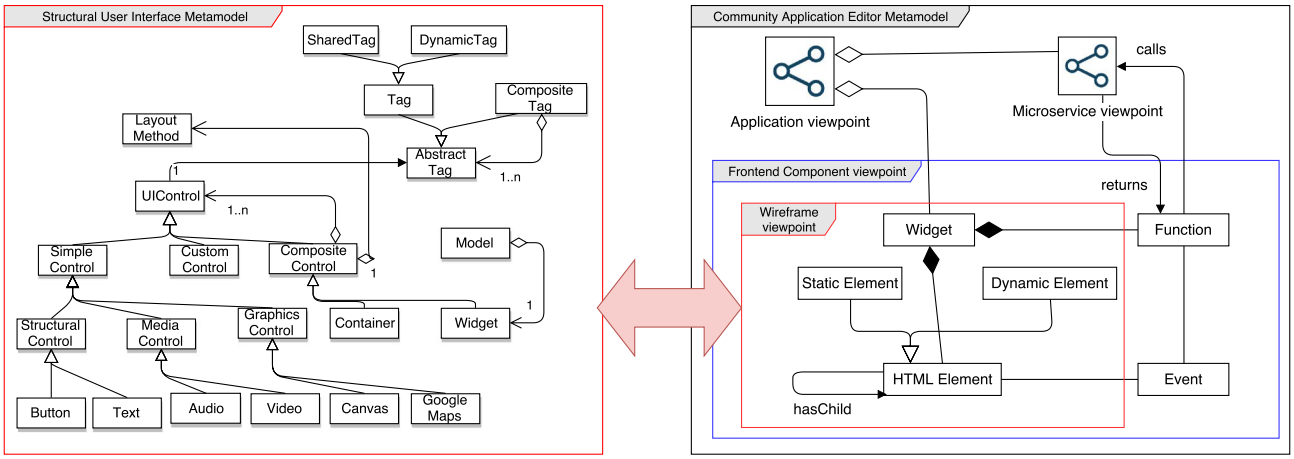


Fig. 6 Mapping of the SUIT- to the MDWE metamodel

Definition 1 A SUIT model is a labeled tree with $SUIT = (V, E)$. V is a finite, non-empty set of vertices. V is always initialized with the root r. E is a set of unordered pairs of distinct vertices $(v1, v2)$ with $v1 \neq v2$, which constitutes the edges of the tree.

$\phi(V) = \{\phi(v) | v \in V\}$ with

$$\phi : V \mapsto V' = (l, t, A) \mapsto (l', t', A')$$

$$: \begin{cases} (l,t,A) \mapsto (l, \text{HTML Element}, & \\ A \cup \{(\text{type}, t), (\text{static}, \text{true}), (\text{collaborative}, \text{false})\}), & \text{for } t \in UI \\ (l, \text{SharedTag}, \emptyset) \mapsto (l', \text{HTML Element}, \mathit{shared}(A')), & \text{for } l' = \mathit{parent}(l) \\ (l, \text{DynamicTag}, \emptyset) \mapsto (l', \text{HTML Element}, \mathit{dynamic}(A')), & \text{for } l' = \mathit{parent}(l) \\ (l, \text{Widget}, A) \mapsto (l, \text{Widget}, A), & \text{otherwise} \end{cases}$$

For the integration into our MDWE approach, a SUIT model is mapped to an instance of the frontend component view. Let $VP = (V, E)$ be an acyclic, directed graph that represents an arbitrary view. An edge $e \in E$ of such a graph has the signature $(l, t, v1, v2, A)$, where $l \in \Sigma$, t is the type of the edge, $v1, v2 \in V$ and A is a set of key value pairs that constitute the attributes of the edge.

Definition 2 An instance M of a view of VP is an acyclic, directed graph with $M = (V', E')$. For each $v \in V'$ holds $\mathit{type}(v) \in \mathit{label}(V)$, with *type* and *label* being helper functions defined as:

$\mathit{type} : V \mapsto \Sigma : (l, t, A) \mapsto t$ and $\mathit{label} : V \mapsto \Sigma : (l, t, A) \mapsto l$.

Analogously, these functions are defined for an edge $e \in E'$ of a view.

Now let $VP_{wireframe}$ be the acyclic directed graph representing an arbitrary instance of the wireframe view and W_{SUIT} a SUIT model representing a concrete wireframe. An instance of the SUIT model is mapped to an instance of the wireframe view with function φ:

$$\varphi : W_{SUIT} \mapsto VP_{wireframe} = (V, E) \mapsto (\phi(V), \gamma(E))$$

where ϕ is defined as follows:

where *shared* and *dynamic* are functions that are applied to every attribute in A of the referenced 'HTML Element' node. These helper functions change the value of the 'collaborative', respectively, 'static' attribute for the referenced 'HTML Element' node. All other attributes are left untouched. Thus, *shared* is defined as

$$\mathit{shared}(A) = \{\mathit{shared}'(a) | a \in A\}$$

with

$$\mathit{shared}' : A \mapsto A : \begin{cases} (k, \mathit{false}) \mapsto (k, \mathit{true}), & \text{for } k = \mathit{collaborative} \\ (k, v) \mapsto (k, v), & \text{otherwise} \end{cases}$$

and *dynamic* is defined as

$$\mathit{dynamic}(A) = \{\mathit{dynamic}'(a) | a \in A\}$$

with

$$dynamic' : A \mapsto A : \begin{cases} (k, true) \mapsto (k, false), & \text{for } k = static \\ (k, v) \mapsto (k, v), & \text{otherwise.} \end{cases}$$

The relationships between the nodes in the wireframe view are generated with γ:

$$\gamma(E) = \{\gamma(e) | e \in E\}$$

with

$$\gamma : E \mapsto E' = (v_1, v_2) \mapsto (l, t, v_1', v_2', A)$$
$$: \begin{cases} (v_1, v_2) \mapsto (l, \text{Wid. To El.}, v_1, v_2, A), & \text{for } v_1 = r, type(v_2) \in UI \\ (v_1, v_2) \mapsto \{(l, \text{hasChild}, v_1, v_2, A)\}, & \text{for } type(v_1) \in UI, type(v_2) \in UI \\ & \text{and } v_1 \neq v_2 \neq r \end{cases}$$

With φ, we only map the UI elements of the SUIT model to the wireframe view. An 'HTML element' node of the frontend component, respectively, wireframe view, consists of the four properties id, type, static and collaborative. The id of the HTML element is automatically generated by the mapping approach. The value of the type attribute is an element from the UI. The static and collaborative attributes are the only attributes represented as tags in the SUIT model. Furthermore, they are simple Boolean attributes and therefore have no own attributes defined. Additionally, the tags are unique and thus they only appear once for a certain UI element.

A node of the SUIT tree is mapped with ϕ to a certain 'HTML Element' or 'Widget' node. An arbitrary UI element of the SUIT model is always mapped to an instance of the 'HTML Element' node class, where the label of the UI element is the label of the node. The type of the UI element is mapped to the type-attribute of the node. By default, the 'static' attribute is true and the 'collaborative' attribute is false. To change the values of these attributes, a DynamicTag, respectively, SharedTag element is mapped to the corresponding attribute in the 'HTML Element' node. For each tag, a function is required, which alters a certain aspect of the signature of an 'HTML Element' node (e.g. type or attribute). For the definition of the current mapping approach, the two helper functions *shared* and *dynamic* are defined, which change the Boolean value of the associated attribute. The root element of the SUIT tree is always mapped to the 'Widget'-node, where the label of the root is also the label of the 'Widget'-node. The same holds for the attributes. With function γ, the relationships between nodes are generated. The function comprises two cases. If the UI element is a direct child from the root, a single 'Widget To HTML Element' edge is created (abbreviated in the function with 'Wid. To El.', due to space restrictions). For the second case, we assume that v_1 is a parent of v_2 and v_1 and v_2 are not the root. Then, the 'hasChild' relationship is generated.

6 Realization

In this section, we present the user interface of the CAE (Sect. 6.1), its general architecture (Sect. 6.2) and the integration of the live code editing (Sect. 6.3) and wireframing (Sect. 6.4).

6.1 User Interface

Figure 7 shows a screenshot of a frontend component modeling space of the CAE. The *Wireframing View* is depicted in the upper right, the *Modeling Canvas* in the lower left and the *Live Code Editor* can be found in the lower right. The upper right depicts the *Live Preview*, the selected modeling element's *Property Window*, the *Persistence Functionality* and on the far right the *Modeling Palette*, *Activity Widget* and *Requirements Bazaar Integration*.

The CAE supports common utility functions, like copy&paste, deletion of an arbitrary number of selected elements and an undo&redo functionality for both the Modeling Canvas, Live Code- and Wireframing Editor. It uses an automatic save functionality. Each altering in the editor saves the current state of all models to the shared editing framework in the Yjs shared data space. Nevertheless, since this is not a persistent storage, the editor offers a persistence functionality, such that all models get stored alongside in a relational database. The editor provides awareness features to support the collaboration. The activity widget shows all collaborators currently working in one of the different views. If a remote user selects one or more elements, each element is highlighted with a surrounding frame and marked with the image of their OpenId Connect profile they used to log into the CAE.

6.2 Architectural Overview

Figure 8 provides an overview of the complete architecture. Our frontend is composed of HTML5 Web components and (apart from the wireframe model, which uses an XML representation) uses JSON representations of the models. We use a lightweight meta-modeling framework, called SyncMeta [13], to realize the collaborative modeling functionalities of

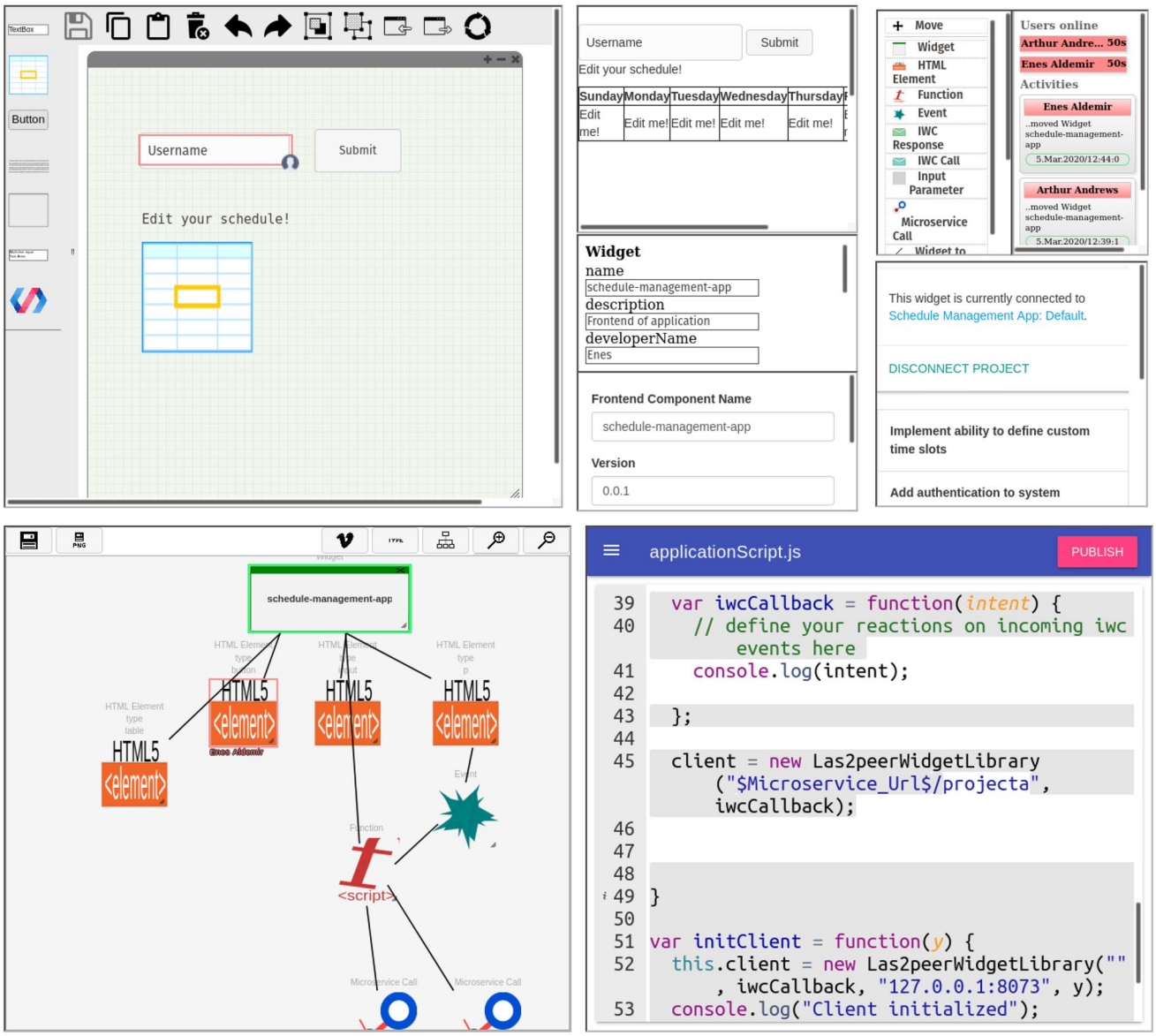


Fig. 7 Screenshot of the CAE

the CAE. It supports NRT collaborative modeling by using Yjs [27], a *Conflict-free Replicated Data Type* (CRDT) framework. For communication with the backend, we use a RESTful interface.

The backend, realized as a las2peer network itself, is composed of two services. The *Model Persistence* service manages the persistence of the microserivce- and frontend component models (together with their enriching wireframe SUIT models, if existing, see Sect. 6.4) in a relational database. The *Code Generation* service implements both the synchronization with the trace models (see Sect. 6.3), as well as it is responsible to generate the resulting source code from the models and trace models. The source code is directly pushed to a GitHub repository, using commit messages to create a history of the modeling process for later reference. We use a Jenkins–Docker continuous deployment pipeline to deploy the resulting services in a las2peer network, directly from the modeling environment.

6.3 Trace Generation and Synchronization

A template engine forms the main component for trace generation and model synchronization. It is used for both the initial code generation, as well as for further model synchronization processes. Figure 9 depicts our trace model, adapted from the metamodel of traces presented in [29].

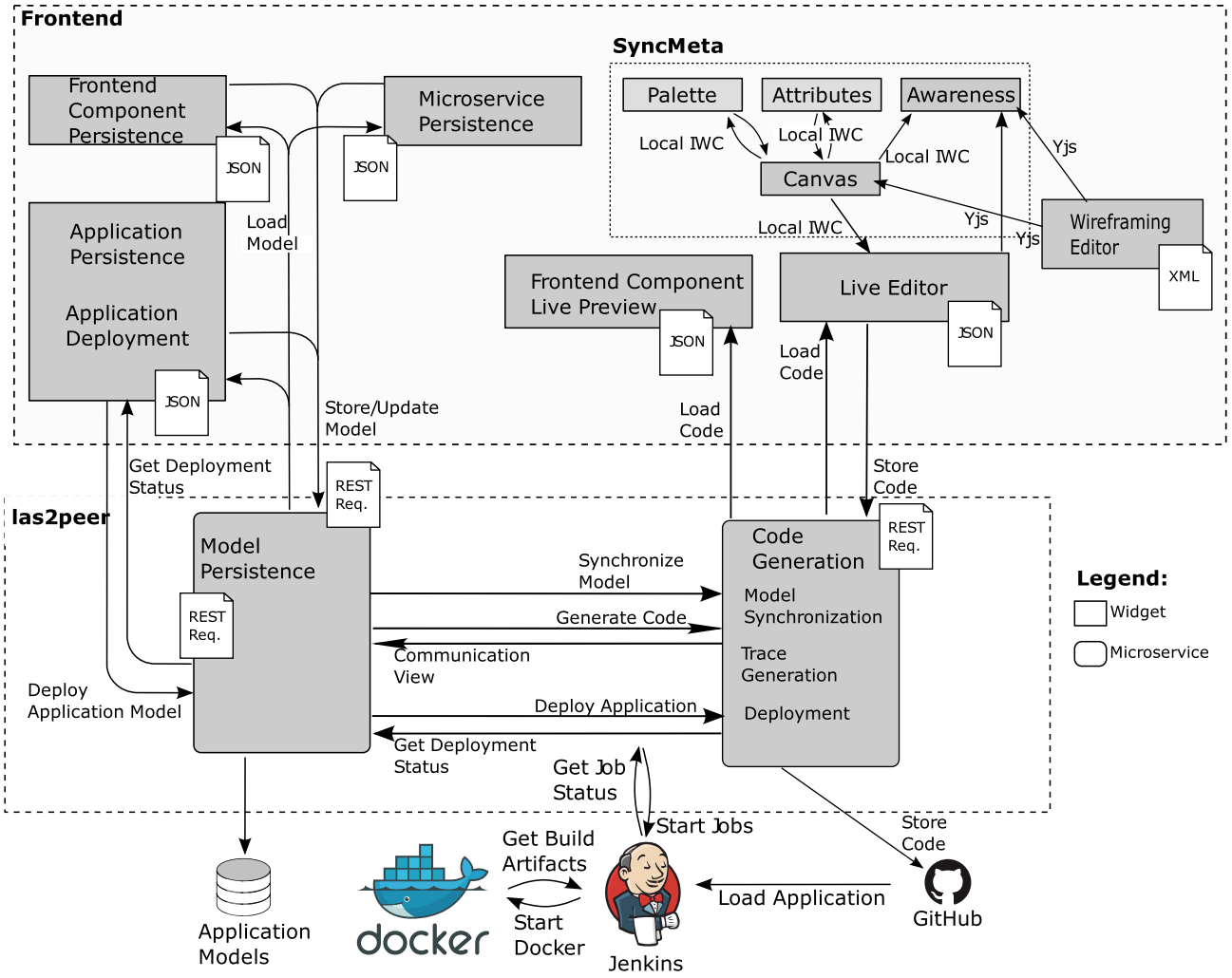


Fig. 8 Architecture of the CAE

For each *FileTraceModel*, and thus for each file, we instantiate a template engine class, which can hold several template objects. A template object is a composition of *Segments*, generated by parsing a template file. A template file contains the basic structure of an element of the Web application's metamodel. In such a file, variables are defined to be used as placeholders, which are later replaced with their final values from the model. Additionally, the template file contains information about which part of the generated source code is protected or unprotected. Based on the template syntax for variables and unprotected parts, a template file is parsed and transformed into a composition of segments of the trace model. For each variable a protected segment is added, and for each unprotected part, an unprotected segment is added to the composition. The parts of a template file that are neither variables nor unprotected parts are also added to the composition as protected segments. According to our previous definition of model synchronization for M2T transformations, Eq. 1 must hold for the model synchronization. Thus, we need to update the content of each variable for all templates of all model elements. However, maintaining a trace and a model element reference for all of these variables is not feasible due to the large size of such a file trace model. Instead, traces are only explicitly maintained for the composition of segments of a template. Linking a segment of a variable to its model element is done implicitly by using the element's id as a prefix for its segment id. When templates are appended to a variable, the type of its linked segment is changed to a composition.

We implemented an initial generation strategy and a synchronization strategy, which are used by our template engine. Each synchronization strategy instance holds a reference to the file trace model of the last synchronized source code to detect new model elements as well as to find source code artifacts of updated model elements. As in some cases, source code artifacts of model elements can

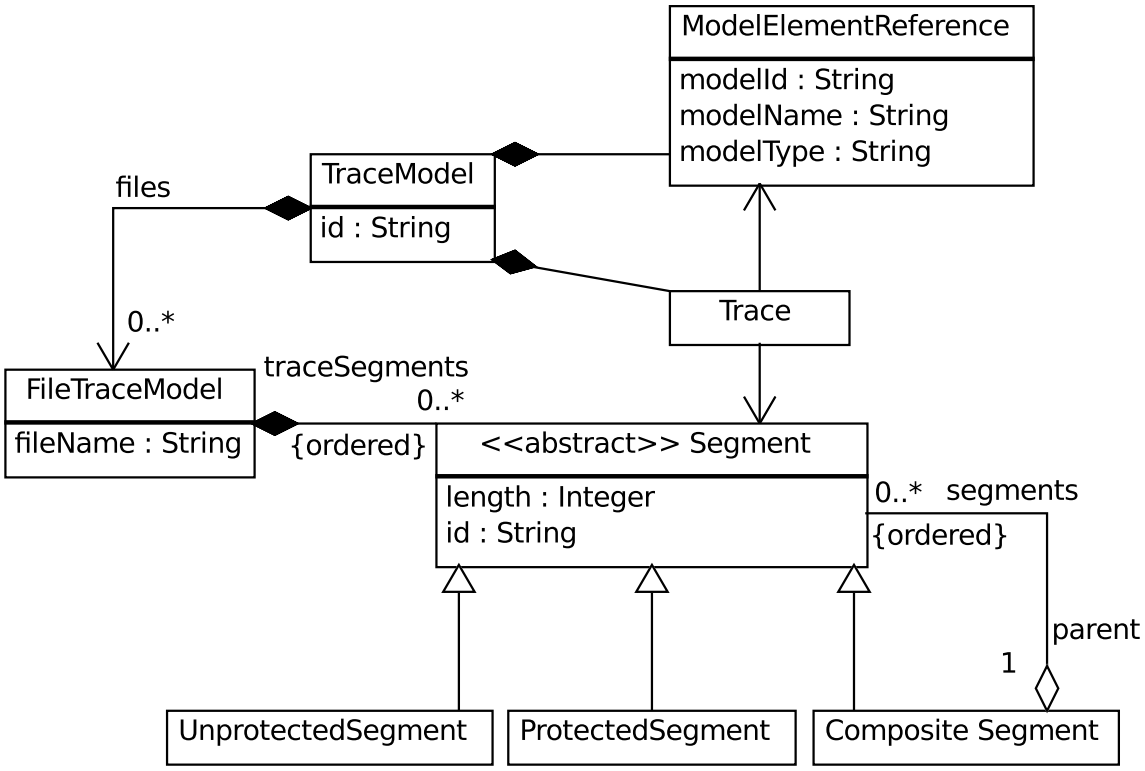


Fig. 9 Trace model used for the live code editing

be located in different files, a synchronization strategy can also hold multiple file trace models in order to find code artifacts across files. After a template engine and its template strategy were properly initiated, the template engine is passed as an argument to the code generators. These create template instances for the model elements based on the template engine. The engine checks, if a segment of the model element is contained in the trace models file by recursively traversing its segments. If a corresponding segment for the model element was found, a template reusing this segment is returned. Otherwise, a new composition of segments, obtained by parsing the template file of the model element, is used for the returned template. For new model elements, new source code artifacts are generated. For updated elements, their corresponding artifacts are reused and updated. As templates can contain other templates in their variables, these nested templates need to be synchronized as well. In the generated final files, source code artifacts of model elements that were deleted in the updated model must be removed from the source code. Therefore, the nested templates, more specifically their segment compositions, are replaced with special segments by the synchronization strategy. These special segments are used as proxies for the original compositions and ensure that templates of deleted model elements are removed from the final source code.

6.4 Wireframe and Frontend Component Model Transformations

Wireframe and frontend component models are persisted next to each other, as the SUIT model enriches the HTML elements of the frontend component model with additional metadata and type-specific attributes. Thus, for code generation, a frontend component model is always required, while the SUIT model is optional. A *Wireframe to Model Transformation* and a *Model to Wireframe Transformation* were developed to transform the SUIT wireframe model to a frontend component model and vice versa. The two transformations are only needed, if one of the two frontend component representations is not existing. After that, the two model states are kept synchronized by the *Live Mapper*.

Wireframe to Model Transformation The wireframe to model transformation takes as input an instance of a wireframe model and the frontend component metamodel. The output of the transformation is a JSON object of the frontend component model. The implementation uses templates of a node, edge, and attribute representation in JSON of the frontend component model. First, the transformation algorithm generates the 'Widget'-node, which represents the root element of the frontend component model. Then, it recursively traverses the wireframe model and creates a corresponding 'HTML Element'-node for each UI control element. The

'type'-attribute of the node is set to the value of the 'HTML Element'-node name of the corresponding UI control element. Furthermore, the 'HTML Element'-node is marked as static and the 'id'-attribute of the node is automatically generated. The value of the id is composed of the 'type'-attribute value and unique. The identifier of the UI control element is reused for the resulting node, which allows to trace back an 'HTML Element'-node to the UI control element. This is necessary for the awareness features and the live mapper. For each node, a 'Widget to HTML Element'-edge is generated, because each node has a connection to the 'Widget'-root node. If the parent of the UI control element is not the root, additionally a 'hasChild'-edge is added to the set of edges. This edge type denotes the hierarchical structure of the wireframe. It connects the parent UI control element to one of its child elements. If a UI control element has the 'shared'-tag assigned to it, the 'collaborative'-attribute of the corresponding 'HTML Element'-node is set to true as well. Since the frontend component metamodel allows every HTML Element to be collaborative, the wireframing editor allows this as well. The result of this transformation is a valid instance of the frontend component metamodel. However, the HTML attributes specified for a certain UI control element are lost, because the frontend component metamodel does not offer a way to represent them. Additionally the width, height and position of the UI control element in the wireframing editor are not related in any way to the position and dimension of corresponding 'HTML Element'-node. Therefore it is necessary to apply an auto-layout for directed graphs to the model, so that it is displayed correctly in the modeling canvas.

Model to Wireframe Transformation The input for this transformation is a JSON representation of the frontend component model and an instance of the wireframe editor. The latter one is required to map the 'type'-attribute of an 'HTML Element' node to the correct UI control element. Since the wireframe only represents the HTML elements of the frontend component model, we only have to consider the 'Widget' node (for the size of the whole frontend component) and those 'HTML Element' nodes that are connected to the 'Widget'-node and marked as static. All other node and edge types of the frontend component model can be ignored for this transformation. As already described in the previous transformation algorithm, certain UI layout information (for example the size and position of elements) is not present in the frontend component model. Thus, we initialize these attributes with default values defined in the wireframe model. Finally, the transformation algorithm assigns the 'shared'-tag to every 'HTML Element'-node which has the 'collaborative'-attribute set to true. The result of the transformation approach is an XML document that represents the wireframe model. The resulting model is then stored in the shared data space alongside with the frontend component model.

Live Mapper The live mapper listens to events of the Modeling Canvas of the frontend component modeling view and to the Wireframing Editor. In contrast to the two previously described transformations, the live mapper directly applies changes to the wireframe and frontend component model and visualizes the results in NRT. Additionally, the live mapper provides awareness features for the selection of entities on both the Modeling Canvas and the Wireframing Editor. To give an example of the live mapping, the creation of a button element in the Wireframing Editor leads to five to six operations on the Modeling Canvas. First, the node is created on the Modeling Canvas, the 'type'-, 'id'-, and 'static'-attributes are set and the new node is connected to the 'Widget'-node. If the button is placed in a container, an additional edge is created between the 'HTML Element'-node representing the container and the new node that represents the button. Furthermore, it is possible to edit the wireframe model through the frontend component model view. For example one can create any UI control element in the Wireframing Editor though the Modeling Canvas by creating an 'HTML Element'-node, connect it to the 'Widget'-node and set the 'static'-attribute to true. After each action on the Wireframing Editor, an auto layout algorithm for directed graphs is applied to the Modeling Canvas, only manipulating those elements that were updated.

7 Evaluation

We evaluated our approach in several iterations, which we describe in this section. For evaluation questionnaires that were filled out at the end of the user evaluation sessions, we used a five-point Likert scale (1–5). For a more extensive coverage of the evaluations of Sects. 7.1–7.5, the corresponding publications mentioned in Sect. 3 can be considered.

7.1 Initial Evaluation

We successfully used the CAE to redesign an existing collaborative Web application used for graph-based storytelling [8]. While this evaluation was only conducted internally, we used it as a first proof-of-concept usage scenario for the CAE.[4] It provided initial feedback on the usability and acted as a first and ongoing test case that lead to several necessary improvements of the framework, before we were able to apply it in the following user evaluations.

[4] https://github.com/wth-acis/CAE-Example-Application.

7.2 Evaluation with Heterogeneous Teams

After we successfully defined the agile and cyclic development approach that builds the basis for development with the CAE, we conducted our first user evaluation [9].

Participants and Procedure We considered groups of two to three people with various technical backgrounds. We carried out 13 sessions, with a total number of 36 participants. The groups consisted of at least one experienced Web developer and at least one member without any technical experience in Web development, who received a description of the application to be designed. During the evaluation session, the non-technical members had to communicate the requirements to the developer team and collaboratively implement the application using the CAE. Each session lasted for about 45 minutes. The goal of this study was to assess the role of NRT collaboration for the development process, and whether our approach improves the integration of non-technical community members into the design and development of Web applications.

Analysis and Outcomes In general, we received high ratings from non-technical members in terms of methodology ("Understanding of separation of concerns" 3.91, "Understanding how application was built" 4.36), and developers felt they were able to implement the requirements formulated by the non-technical members (4.64). Most non-technical members felt integrated well into the NRT development process (4.27) and the oral interviews revealed that they could follow the development process well. Although the question, if non-technical members took an active role in the development process received the lowest score, the result is still pretty high (3.82). From the developer survey, we received the highest ratings for questions regarding the concept of CAE and its usability ("Understanding functionality" 4.79, "Understanding separation of concerns" 4.71). Collaborative aspects were also rated rather high by both groups. The oral interviews revealed that most developers felt both the need for requirement analysis improvements regarding the inclusion of non-technical stakeholders as well as that the CAE can be used for this purpose.

The evaluation showed the usefulness of the CAE to integrate non-technical members better into the development process. Developers saw the benefit of CAE's MDWE approach to contribute to a unified community application landscape. A particularly often requested feature was the introduction of a second abstraction tier for the frontend component view, which could hide too technical aspects from non-technical members, concentrating more on the "visible" elements, putting the functionality into a second component view, which would then be used by the developers only. Another issue mentioned by the developers was the need to adjust the generated source code to fully reflect the requirements, and then having no possibility to return to the modeling environment, since the modified code would be overwritten when the code was regenerated by the framework. We used this feedback to start developing the Live Code Editor, as well as the Wireframing Editor, which both tackle this problem from different directions, but both use the same idea of providing different views on the same model.

7.3 Evaluation in a Lab Course

While we were developing the aforementioned live coding and wireframing extensions, we in parallel started to validate our MDWE process with its modeling and development phases over a longer period of time, by studying the impact it has on Web developers [9, 10]. Therefore, it was necessary to extend the CAE with automated deployment features, such that the created applications could be used in practice, to validate their functionality.

Participants and Procedure We evaluated our approach in a lab course of 15 undergraduate computer science students. The students had basic programming knowledge, in particular in Java (4.6) from their first programming lectures, but our pre-survey also indicated that none of them were really familiar with Web development (1.67) or microservice architectures (1.73). During a two week period, the students were asked to model and deploy the basic framework of their lab course prototype.

Analysis and Outcomes In this evaluation, we were especially interested in how the CAE can help developers that are not yet familiar with the present development environment. Our questionnaire thus focused on the learning effects the CAE has on developers that have to integrate into a new development process. Our results indicate a high learning effect in terms of understanding the underlying Web development concepts of microservices (4.43 vs 1.73) and frontend components (4.5 vs 2.6). We received rather high ratings in terms of MDWE easing the learning of new concepts and techniques (3.86) and MDWE improving the understanding of the generated application (3.71).

Occurring problems during this evaluation were mainly due to the use of an experimental prototype which was never tested in an environment with more than a handful of people using it at the same time. Boundary conditions and network latency problems lead to a cycle of fixes, version incompatibilities and newly introduced problems. Even though this might have clouded the participants' impression of CAE use, it finally lead to major technical improvements of our framework. These first results of a more realistic usage setting showed promising applications of the CAE as a tool to teach developers of different domains the development of Web applications with a p2p microservice architecture.

7.4 Live Code Editor Evaluation

Due to the feedback received in the previous two evaluations, we developed the Live Code Editor. On this, we performed a usability study with student developers to assess how it integrates into our collaborative MDWE methodology and how well it performs and is received in practice [12].

Participants and Procedure We carried out eight user evaluation sessions, each consisting of two participants. After receiving a short introduction and filling out a pre-survey to asses their experiences in Web development, the participants were seated in the same room and asked to extend an existing application, which consisted of two frontend components and two corresponding microservices. As expected, the pre-survey rating of the familiarity with Web technologies (4.00) was rather high. However, only a minority of our participants were familiar with MDWE (2.67) or had used collaborative coding for creating Web applications before (2.40). Each evaluation session took about 30 minutes of development time.

Analysis and Outcomes The participants rated connections between our two collaborative phases, namely the access to the code editor from the model (4.67) and the reverse process with the synchronization enabled (4.40) very high. Even though the chosen application was, due to the time constraints of a live evaluation setting, quite simple, the evaluation participants mostly saw cyclic development in general as relevant (4.13) and also rated the benefits of a cyclic MDWE process high (4.00). Moreover, all participants identified the advantages of code and model synchronization (4.33). We considered the results of this evaluation as a proof-of-concept, that the Live Code Editor technically fulfills its purpose as a way to integrate live coding into the cyclic development, mitigating the need to manually change the generated source code and thereby break the MDWE cycle.

7.5 Wireframing User Evaluation

After we successfully developed and integrated the Live Code Editor, we tackled the feedback gained in our evaluation with heterogeneous teams (see Sect. 7.2) and developed the Wireframing Editor. We then evaluated it to gain user feedback on how well it integrates into the process and how it changes the way applications are developed with the CAE [11].

Participants and Procedure We recruited eight student developers as participants, which were split up into groups of two, resulting in four evaluation sessions that each lasted about 60 minutes. As in the Live Code Editor evaluation, the pre-survey revealed a high familiarity with Web development (4.50). Also, a rather high familiarity with MDWE concepts was observed (3.13), but wireframing editors were not very familiar to the participants (2.75). The participants were asked to develop a frontend for an already existing microservice backend. A specification for both the existing RESTful API, as well as the desired Web frontend was handed out to the participants at the beginning of the session.

Analysis and Outcomes With an average of 4.00, most participants found the wireframing editor was easy to use and with an average of 4.13 and 4.25, the participants found both their application reflected in the live preview widget as it was designed by them, as well as that they were aware of what their collaborator did in the wireframing editor. Also, participants mostly agreed that the modeling canvas and the wireframing view reflected the same model (4.25). With an average rating of 4.25, the majority of the participants thought the wireframing editor a useful extension for MDWE frontend development and the integration of the wireframe into the process was understood quite well (4.38).

7.6 Activity Evaluation

To gain a deeper understanding about the collaboration process and working behavior of the participants, we monitored the participants' activities in the CAE.

Participants and Procedure We monitored five sessions with two participants each. Each time a participant switched the widget, the time spend in the widget was logged. Furthermore, if a participant altered the wireframe, model or code, an additional event was logged. For each participant the time spend in each editor was aggregated and the total amount of altering activities a participant issued was counted.

Analysis and Outcomes Table 1 depicts for each participant the relative time spend in a certain widget, as well as the number of absolute activities in this widget. A particular activity can be a create-, move-, resize-, delete- or attribute change event of an element in the Wireframing Editor or Modeling Canvas, or a value change activity in the Live Code Editor.

The results clearly indicate that the participants spend the most time in the Modeling Canvas. One explanation for this might be that the participants had to get familiar with the modeling language first, which corresponds with our observations, that during the first few minutes participants did not use it productively, but experimented with different modeling elements, until they were familiar with them. With an average of $AVG_{model} = 188$ activities, the modeling part was also the most demanding task. With an average of $AVG_{code} = 13$ activities the coding task was less work intensive. One reason for that might have been that all participants had some development experiences. The average number of activities to complete the wireframing task was quite low with $AVG_{wireframe} = 10$, and also the time spend in the Wireframing Editor was quite short, compared to the time spend in the Modeling Canvas. The participants of the

Table 1 Results of the activity evaluation for each session and participant

	Participant 1			Participant 2		
	Wireframe	Model	Code	Wireframe	Model	Code
Session 1	(20.1%, 20)	(26.1%, 180)	(53.1%, 14)	(38.9%, 18)	(27.4%, 87)	(33.7%, 2)
Session 2	(7%, 3)	(81.3%, 104)	(11.7%, 4)	(15.4%, 10)	(41.7%, 133)	(42.9%, 15)
Session 3	(7.1%, 12)	(79.6%, 291)	(13.4%, 20)	(17.6%, 3)	(65%, 288)	(17.4%, 6)
Session 4	(11.9%, 0)	(51.5%, 169)	(36.6%, 11)	(9.4%, 8)	(57.7%, 116)	(32.9%, 5)
Session 5	(53.8%, 20)	(44.6%, 61)	(1.6%, 0)	(34.8%, 19)	(56.1%, 78)	(9.1%, 0)

The relative time spend in each widget, as well as the absolute number of activities for each widget is given

fifth session were not able to generate the code, which is why there is almost no activity and usage time recorded in the Live Code Editor.

These results indicate that the Wireframing Editor was easier to use and required less time to get familiar with, which was a major goal and a key requirement of the editor. Nevertheless it has to be mentioned, that the evaluation of time spend in each editor and activity monitoring is tightly coupled with the evaluation task, and thus influenced by it.

7.7 Service Success Measurement Evaluation

Our final evaluation concerned the integration of the CAE in the overarching las2peer methodology (see also Sect. 4.3), with a special focus on the Requirements Bazaar integration and the usage of service success measurement modeling elements, which then were used for visualization in las2peer's monitoring and evaluation suite.

Participants and Procedure We recruited thirteen participants from a universities computer science department and conducted thirteen evaluation sessions. Participants were given an existing Web service for image uploading, which had a certain, yet obvious, flaw It contained an (artificial) delay in the upload process. The service itself was already developed with the CAE and a corresponding Requirements Bazaar project existed, that already documented the flaw. The workflow of the evaluation contained first reading the documentation of the issue in the Requirements Bazaar, and then reacting by first measuring the image uploading time via a modeled monitoring message extension and finally correcting it in the live code editor. The resulting improvement could then also visually be confirmed in las2peer's monitoring and evaluation suite.

Analysis and Outcomes While the evaluation was concerned with several aspects of the monitoring and evaluation suite, three questions were posed to specifically justify the Requirements Bazaar and monitoring integration into the CAE. With an average score of 4.32, most participants found the Requirements Bazaar well integrated into the CAE. Our observation was, that participants had no problems browsing the requirements connected to the modeling project directly from the CAE's interface. Another question to verify this was, if participants were able to distinguish the responsibilities of the CAE, the Requirements Bazaar and las2peer's evaluation suite. This was answered with an average score of 3.83, which, taking into account the short time the participants had to get familiar with the concept, is a clear sign that the responsibilities of the individual components were understood. Finally, we asked, if the combination of the three components was perceived as a way to make monitoring features more transparent to non-technical stakeholders, which was answered with an average score of 3.71. Taking these results together, we perceive the CAE well integrated into the overarching methodology.

8 Conclusions and Future Work

In this contribution, we presented the Community Application Editor, a collaborative, view-based modeling approach for microservice-based Web applications. It integrates live code editing and wireframing. We formally defined this integration by specifying the needed transformations and synchronizations. By following a design science approach, we cyclically developed and evaluated our prototype and integrated it into an overarching methodology for developing and distributing community microservices in an open source, decentralized p2p infrastructure. In the final steps of the research carried out in the scope of this contribution, we coupled the editor more tightly with this methodology by introducing monitoring capabilities and requirements elicitation directly from within the modeling environment. The results promise, that a formalized way of developing microservice-based architectures with the possibilities for developers to use live code editing and the integration of wireframing to provide an additional view on the frontend, can advance Web application engineering and contribute to agile practices and rapid prototyping. In future work, we plan to evaluate the approach further in larger development projects.

Compliance with ethical standards

Conflict of Interest The authors declare that they have no conflict of interest.

References

1. Angelaccio M (2016) MetaPage—a data intensive MockupDD for Agile web engineering. In: WEBIST 2016. INSTICC, SciTePress. https://doi.org/10.5220/0005927103150317
2. Angyal L, Lengyel L, Charaf H (2008) A synchronizing technique for syntactic model-code round-trip engineering. In: Proceedings of the 15th international conference and workshop on the engineering of computer-based systems, ECBS '08. IEEE Computer Society, pp 463–472. https://doi.org/10.1109/ECBS.2008.33
3. Arnowitz J, Arent M, Berger N (2010) Effective prototyping for software makers. Interactive technologies. Elsevier Science, Amsterdam
4. Brambilla M, Fraternali P (2014) Interaction flow modeling language: model-driven UI engineering of web and mobile apps with IFML. The MK/OMG Press, Morgan Kaufmann
5. Ceri S, Fraternali P, Bongio A (2000) Web Modeling Language (WebML): a modeling language for designing web sites. Comput Netw 33(1):137–157
6. Czarnecki K, Helsen S (2006) Feature-based survey of model transformation approaches. IBM Syst J 45(3):621–645. https://doi.org/10.1147/sj.453.0621
7. de Lange P, Goschlberger B, Farrell T, Neumann AT, Klamma R (2020) Decentralized learning infrastructures for community knowledge building. IEEE Trans Learn Technol. https://doi.org/10.1109/TLT.2019.2963384
8. de Lange P, Nicolaescu P, Derntl M, Jarke M, Klamma R (2016) Community application editor: collaborative near real-time modeling and composition of microservice-based web applications. In: Modellierung 2016 workshop proceedings, pp. 123–127
9. de Lange P, Nicolaescu P, Klamma R, Jarke M (2017) Engineering web applications using real-time collaborative modeling. In: Collaboration and technology: 23rd international conference, CRIWG 2017, Saskatoon, SK, Canada, August 9–11, 2017, Proceedings. Springer International Publishing, pp 213–228. https://doi.org/10.1007/978-3-319-63874-4_16
10. de Lange P, Nicolaescu P, Klamma R, Koren I (2016) DevOpsUse for rapid training of agile practices within undergraduate and startup communities. In: Adaptive and adaptable learning, *LNCS*, vol 9891. Springer International Publishing, pp 570–574. https://doi.org/10.1007/978-3-319-45153-4_65
11. de Lange P, Nicolaescu P, Rosenstengel M, Klamma R (2019) Collaborative wireframing for model-driven web engineering. In: Web information systems engineering—WISE 2019, LNCS, vol 11881. Springer Nature, pp. 373–388. https://doi.org/10.1007/978-3-030-34223-4_24
12. de Lange P, Nicolaescu P, Winkler T, Klamma R (2018) Enhancing model-driven web engineering with collaborative live coding. In: Modellierung 2018
13. Derntl M, Nicolaescu P, Erdtmann S, Klamma R, Jarke M (2015) Near real-time collaborative conceptual modeling on the web. In: 34th international conference on conceptual modeling (ER 2015), LCNS, vol 9381. Springer International Publishing, pp 344–357. https://doi.org/10.1007/978-3-319-25264-3_25
14. Giese H, Wagner R (2006) Incremental model synchronization with triple graph grammars. In: Proceedings of the international conference on model driven engineering languages and systems. Springer, pp 543–557. https://doi.org/10.1007/11880240_38
15. Gotel O, Cleland-Huang J, Hayes J.H, Zisman A, Egyed A, Grünbacher P, Dekhtyar A, Antoniol G, Maletic J (2012) The grand challenge of traceability (v1.0). In: Software and systems traceability. Springer, pp 343–409. https://doi.org/10.1007/978-1-4471-2239-5_16
16. Hettel T, Lawley M, Raymond K (2008) Model synchronisation: definitions for round-trip engineering. In: Proceedings of the first international conference on model transformations. Springer, pp 31–45. https://doi.org/10.1007/978-3-540-69927-9_3
17. Hevner AR, March ST, Park J, Ram S (2004) Design science in information systems research. MIS Q 28(1):75–105
18. Kent S (2002) Model-driven engineering. In: International conference on integrated formal methods, pp 286–298
19. Klamma R, Renzel D, de Lange P, Janßen H (2016) las2peer—a primer. https://doi.org/10.13140/RG.2.2.31456.48645
20. Koch N, Kraus A (2002) The expressive power of UML-based web engineering. In: Second international workshop on web-oriented software technology (IWWOST02), vol 16, pp 105–119
21. Koch N, Meliá-Beigbeder S, Moreno-Vergara N, Pelechano-Ferragud V, Sánchez-Figueroa F, Vara-Mesa JM (2008) Model-driven web engineering
22. Koren I, Klamma R, Jarke M (2020) Direwolf model academy: an extensible collaborative modeling framework on the web. In: Modellierung 2020 workshop proceedings. GI, pp 213–216
23. Lankhorst M (2013) Enterprise architecture at work: modelling, communication and analysis, 3rd edn. The enterprise engineering series. Springer, Berlin
24. Mellor S.J, Scott K, Uhl A, Weise D (2002) Model-driven architecture. In: Proceedings of the 8th international conference on object-oriented information systems. Springer, pp 290–297. https://doi.org/10.1007/3-540-46105-1_33
25. Mens T, van Gorp P (2006) A taxonomy of model transformation. Electron Notes Theor Comput Sci 152:125–142. https://doi.org/10.1016/j.entcs.2005.10.021
26. Nicolaescu P, Rosenstengel M, Derntl M, Klamma R, Jarke M (2018) Near real-time collaborative modeling for view-based web information systems engineering. Inf Syst 74:23–39
27. Nicolaescu P, Jahns K, Derntl M, Klamma R (2016) Near real-time peer-to-peer shared editing on extensible data types. In: GROUP 2016. ACM. https://doi.org/10.1145/2957276.2957310
28. Ogunyomi B, Rose LM, Kolovos DS (2014) On the use of signatures for source incremental model-to-text transformation. In: 17th international conference on model driven engineering languages and systems, MoDELS '14. Springer International Publishing, pp 84–98. https://doi.org/10.1007/978-3-319-11653-2_6
29. Olsen GK, Oldevik J (2007) Scenarios of traceability in model to text transformations. In: Proceedings of the third European conference on modelling foundations and applications, ECMDA-FA '07. Springer, pp 144–156. https://doi.org/10.1007/978-3-540-72901-3_11
30. Peffers K, Tuunanen T, Rothenberger MA, Chatterjee S (2007) A design science research methodology for information systems

research. J Manag Inf Syst 24(3):45–77. https://doi.org/10.2753/MIS0742-1222240302
31. Renzel D, Behrendt M, Klamma R, Jarke M (2013) Requirements bazaar: social requirements engineering for community-driven innovation. In: 21st IEEE international requirements engineering conference: RE 2013. IEEE, pp 326–327. https://doi.org/10.1109/RE.2013.6636738
32. Renzel D, Klamma R, Jarke M (2015) IS success awareness in community-oriented design science research. In: New horizons in design science: broadening the research agenda, LNCS, vol 9073. Springer International Publishing, pp 413–420. https://doi.org/10.1007/978-3-319-18714-3_33
33. Rivero J.M, Grigera J, Rossi G, Robles Luna E, Koch N (2012) Towards Agile model-driven web engineering. In: IS olympics: information systems in a diverse world: CAiSE forum 2011, London, UK, June 20–24, 2011, selected extended papers. Springer, Berlin, pp 142–155. https://doi.org/10.1007/978-3-642-29749-6_10
34. Rivero J.M, Rossi G (2013) MockupDD: facilitating agile support for model-driven web engineering. In: IWCE 2013 workshops. Springer International Publishing, pp 325–329. https://doi.org/10.1007/978-3-319-04244-2_31
35. Rivero J, Rossi G, Grigera J, Luna ER, Navarro A (2011) From interface mockups to web application models. In: International conference on web information systems engineering, pp 257–264
36. Schwabe D, Rossi G (1998) An object oriented approach to web-based applications design. TAPOS 4(4):207–225
37. Vara JM, Bollati VA, Jimenez A, Marcos E (2014) Dealing with traceability in the MDD of model transformations. IEEE Trans Softw Eng 40(6):555–583. https://doi.org/10.1109/TSE.2014.2316132