I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, C. Seidl (Hrsg.): Modellierung 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 199

Enhancing MDWE with Collaborative Live Coding

Peter de Lange, Petru Nicolaescu, Thomas Winkler, Ralf Klamma[1]

Abstract: Model-Driven Web Engineering (MDWE) methodologies enhance productivity and offer a high level view on software artifacts. Coming from classical software development processes, many existing approaches rather enforce a top-down structure instead of supporting a cyclic approach that integrates smoother with modern agile development. State-of-the-art MDWE should integrate established and emerging Web development features, such as (near real-time) collaborative modeling and shared editing on the generated code. The challenge when covering these requirements lies with synchronizing source code and models, an essential need to cope with regular changes in the software architecture and provide the flexibility needed for agile MDWE. In this paper, we present an approach that enables cyclic, collaborative development of Web applications by using traceability in model-to-text transformations to deal with the synchronization. We adopt a trace-based solution for collaborative live coding in order to merge manual code changes into Web application models and ensure that the open-source code repositories reflect both model and manual code refinements. Our evaluation shows a reliable code to model synchronization and investigates the usability in collaborative software development settings. With our approach we contribute to integrating agile development practices into MDWE.

Keywords: Model-Driven Web Engineering; Traceability; Model to Text Transformations; Collaborative Live Coding

1 Introduction

Mostly adopted in classical software development models, past MDWE research does not cope with the paradigm shift towards agile development [MR03], inclusion of end-users and various stakeholders into the development process and the increased communication and collaboration in (remote) teams on the Web. To adapt to this new intensive information exchange setting, a MDWE approach has to support development cycles with rapid changes in the architecture and code being simultaneously edited, all in a multi-user collaborative and *Near Real-Time* (NRT) scenario. Hence, traditional methods that enable the synchronization between model and code need to be adapted to the collaborative setting. Furthermore, they need to cope with powerful, collaborative frontend technologies and paradigms beyond simple website and client-server models.

In this paper, we present a cyclic MDWE development process in which model updates are synchronized with source code refinements. The authoring of models and code is

[1] RWTH Aachen University, Lehrstuhl Informatik 5, Advanced Information Systems Group (ACIS), Ahornstrasse 55, 52074 Aachen, Germany {lastname}@dbis.rwth-aachen.de

performed collaboratively in NRT on the Web, bringing together teams composed of various stakeholders. We apply related work on traceability [OO07] and synchronization [HLR08] from the model-driven engineering domain to formalize an agile collaborative MDWE method for the Web. In order to adapt the synchronization techniques to the NRT collaboration setting, we developed a trace model that provides linking information between model elements and source code artifacts. We apply this concept using a prototype that integrates a live collaborative code editor into an existing MDWE framework. All source code and traces are stored in a source code repository using the Git protocol.

In the following, we start with introducing the background and related work our paper is based on in Section 2, before Section 3 introduces our MDWE process. Section 4 presents the formalization of our traceability-based synchronization approach. Section 5 describes the integration of our proof-of-concept prototype into the CAE [La17], a Web-based MDWE framework. In Section 6, we describe and interpret the results of our user evaluation. Finally, Section 7 concludes this paper and provides an outlook on future work.

2 Background and Related Work

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 [Mv06], 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 models' metamodel [Me02]. Template-based approaches are (together with visitor-based approaches) the most prominent solution for M2T transformations [CH06]. 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 [ORK14]. 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 [ALC08]. Another challenge is the decision on the appropriate granularity of traces, as the more detailed the links are, the more error-prone they become [Go12, Va14]. In addition to model synchronization, *Round-Trip Engineering* (RTE) also considers changes in the source code which are propagated back into the model. Among others, formal definitions of model synchronization and RTE for M2M transformations have been proposed in [GW06] and [HLR08].

OOHDM was one of the first approaches towards the changing requirements in the development of Web applications in comparison to traditional applications [SR98]. It focuses on the hypertext structure of Web applications, as traditional software engineering concepts do not offer appropriate abstractions for those structures. Following the *separation of concerns* approach, OOHDM divides the development process into four tasks, i.e. the

conceptual design, the navigation design, the abstract interface design and the implementation. In *UML-based Web Engineering (UWE)*, the modeling of Web applications integrates the separation of concerns by separately modeling content, navigation, business processes and presentation [KKK07]. While stereotypes are used to define specific semantics of model elements, e.g. "navigationLink" for direct links, the *Object Constraint Language* (OCL) is used to define additional static semantics and constraints for a model element such as class invariants. *MagicUWE* is a plugin for the commercial CASE tool MagicDraw that supports the UWE notation [BK09]. Another prominent modeling language is *WebML*, in which Web applications are defined by high-level and platform-independent specifications [CFB00]. While a structural model describes the site content, a hypertext model is responsible for the composition of contents and the navigation between pages. In a presentation model, the layout and graphical representation of a page is defined independently of the displaying device and the language used for the visual representation. In addition, a personalization model is used to store user specific content [CFB00]. In 2013, an extension of WebML lead to the specification of the *Interaction Flow Modeling Language* (IFML), a language that was adopted as a standard by the *Object Management Group* (OMG). While especially UWE and WebML are the most prominent examples of recent developments in the domain of MDWE research, none of them are based on a (formalized) RTE approach that allows for an agile use of these approaches.

Medini QVT, developed by IKV++, is a commercial tool that implements the declarative part of the QVT specification, i.e. the QVT Relations. Thus, it supports incremental bidirectional M2M transformations and the generation of trace models during the transformation process. *UML Lab*, developed by Yatta, is a commercial modeling suite which is integrated into the Eclipse platform. It provides a graphical UML modeling editor and template-based M2T transformations, supporting reverse engineering and RTE. However, NRT collaboration facilities are not provided. Finally, *MOFScript* is a M2T transformation tool developed as an Eclipse plugin. As the MOFScript language does not depend on any actual metamodel, it can be used for code generation of arbitrary metamodels and their instances. In addition, it supports the generation of traces as well as the synchronization between models and source code. However, it does not provide any RTE facilities. Besides, the tool does neither offer NRT collaborative modeling nor coding functionalities. To our knowledge, there currently exists no agile NRT MDWE framework, that allows for a cyclic, collaborative development process with modeling phases followed by (live) coding phases and vice-versa.

3 Agile Collaborative MDWE

Our approach introduces an agile life-cycle for MDWE. From a general perspective on the modeling-coding cycle, changes in the architecture (i.e. changes happening in NRT as a team work result of multiple stakeholders with and without technical knowledge) are performed in the collaborative modeling phase. Detailed behavior is refined in the collaborative coding phase, using the automatically generated code from the model artifacts.

The cycle is achieved by synchronizing collaborative modeling phases and live source code editing. At any point in time, stakeholders can switch between one of the two phases. All changes of the model are immediately reflected in the generated source code. Changes to the source code are taken into account upon model-to-code regeneration, integrating them accordingly into the regenerated source code. Both modeling and coding is done on the Web in a NRT collaborative manner, meaning that changes in model and source code are directly visible to all stakeholders at all time.

Although our approach can be used for arbitrary MDWE frameworks and Web applications, in the scope of this work we consider Web applications composed of HTML5 and JavaScript frontends and RESTful microservice backends. Since especially frontend architectures can be highly unstructured, we propose to unify the architecture of applications developed with our approach through the usage of protected segments, that enforce a certain base architecture, facilitating both 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.

4 Model Synchronization Strategy

Although the conceptual idea of our model synchronization strategy can be applied to arbitrary Web application metamodels, for a better understanding we give simple examples of the appliance of our concept to the metamodel we use in our implementation at certain points in this section. Therefore, in Fig. 1 we illustrate an excerpt of our *Frontend Component* metamodel. It contains the three elements *HTML Element*, *Event* and *Function*, their attributes and their connections between each other.

Our general concept of model synchronization is depicted in Fig. 2. It 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

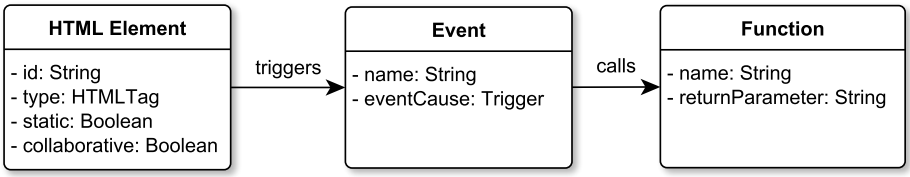

Fig. 1: Example model

synchronization expressed in [HLR08] 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 $trans : M_A \rightarrow M_B$ and a function $strip : M \times (M_A \rightarrow M_B) \rightarrow M$ 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 [HLR08]. 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. As an example, consider the *height* attribute of an HTML *img* tag. As it can be seen in Fig. 1, the *HTML Element* of our metamodel does not contain a height attribute, so this manually added attribute would not be part of the stripped model according to our transformation.

4.1 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. 2, 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 i-th code refinement phase. Formally, a single source code change can be denoted by one of the two functions $\delta^+_{M_T} : M_T \times C \times \mathbb{N} \rightarrow M_T$ and $\delta^-_{M_T} : M_T \times C \times \mathbb{N} \rightarrow M_T$. 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:

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

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

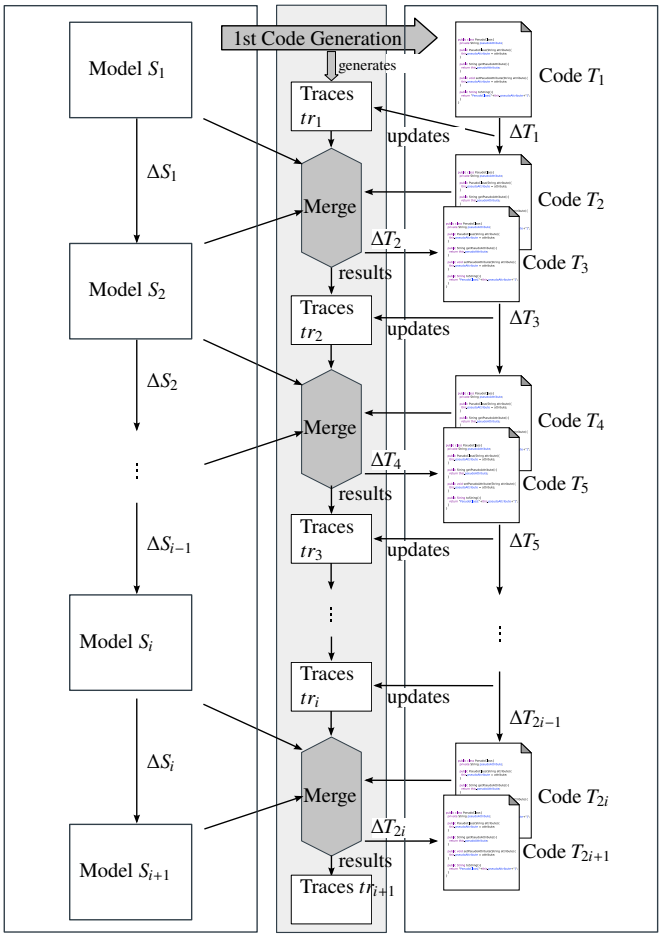

Fig. 2: Model synchronization

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:

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

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

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

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:

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

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:

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

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

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

where $l_i \in \mathbb{N}$ for $i, m \in \mathbb{N}, 1 \leq i \leq m$ is the length of the i-th 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.

4.2 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 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 [HLR08]: 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 $\delta^{attr}_{a,s1,v}$: 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.

As the transformation of model to source code changes is highly dependent on the type of the updated model elements, the concept for synchronization is shown exemplary for *Events* of the example model described in Fig 1. A valid *Event* element has two edges e and e' that connect it to an *HTML Element* h and to a *Function* element f, respectively. Then, a newly created *Event* element is transformed to source code changes by

$$\delta^{+}_{t}(S_i) \rightarrow \delta^{*+}(T_{2i}, t_{event}, pos_{events}) \tag{17}$$

where pos_{events} references the position in the source code that contains all events and t_{event} is the following template:

$$t_{event} := temp_t(attr_{`name`}(event), attr_{`cause`}(event), attr_{`name`}(f), attr_{`id`}(h)) \tag{18}$$

Thereby, a new source code artifact representing the *Event* element is inserted into the source code. The deletion of an *Event* element is transformed as follows:

$$\delta^{-}_{t}(S_i) \rightarrow \delta^{*-}(T_{2i}, pos_{seg}(event), len_{seg}(event)) \tag{19}$$

The code artifact of the deleted *Event* element is removed from the source code. Finally a value of an attribute a is transformed to source code changes:

$$\delta^{attr}_{a,event,v}(S_i) \rightarrow \delta^{*+}(\delta^{*-}(T_{2i}, pos_{attr}(a, event), len_{attr}(a, event)), v, pos_{attr}(a, event)) \tag{20}$$

Thus, the old value of the attribute is first deleted from the source code and its new value is inserted at the same position. As we are only considering valid models and according to our model, an *Event* element needs two edges, we can assume that for each deletion of an edge there is also an insertion of a new edge. Therefore, we only transform edge updates. Since an *Event* element has two edges, we need to differentiate:

1. If the current *HTML Element h* connected to an *Event* is changed to another *HTML Element h′*, the transformation from model to source code is:
$$\delta^{+}_{e,event,h'}(\delta^{-}_{e,event,h}(S_i)) \rightarrow \delta^{*+}(\delta^{*-}(T_{2i}, pos_{attr}(\text{`}id\text{`}, h), len_{attr}(\text{`}id\text{`}, h)), attr_{\text{`}id\text{`}}(h'), pos_{attr}(\text{`}id\text{`}, h))$$
2. If the current *Function* element f of an *Event* is changed to another *Function* element f', the source code is modified according to:
$$\delta^{+}_{e,event,f'}(\delta^{-}_{e,event,f}(S_i)) \rightarrow \delta^{*+}(\delta^{*-}(T_{2i}, pos_{attr}(\text{`}name\text{`}, f), len_{attr}(\text{`}name\text{`}, f)), attr_{\text{`}name\text{`}}(f'), pos_{attr}(\text{`}name\text{`}, f))$$

5 Realization

We integrated our model synchronization strategy into a Web-based collaborative modeling environment called *Community Application Editor* (CAE) [La16]. CAE uses a template-based MDWE approach to support NRT collaboration between developers and other involved stakeholders. In the scope of this paper's contribution, we extended the frontend with a *Live Code Editor* widget based on the *ACE editor*[2], which realizes the collaborative NRT editing of (generated) source code directly in the browser. The NRT collaboration and shared editing features are realized using the Yjs [Ni16] framework. Available as an open source Javascript library, Yjs enables shared editing for arbitrary data types and formats (e.g. lists, maps, objects, text, JSON) on the Web. It uses protocols such as WebRTC and WebSockets for message propagation in NRT. Integrated with the live code editor widget, our framework only needs to manage the Yjs *Collaboration Spaces* (similar to a chat room, with all involved users being able to collaborate on one application model and code). We integrated the model synchronization and trace generation functionality into the Java-based backend of the CAE and extended it with means to manage local Git repositories used by the live code editor widget to update the source code.

[2] `https://ace.c9.io`

Trace Generation and Model Synchronization The template engine, implemented in the backend of the CAE, 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. Except for some special cases, like renaming or deleting files, applying the *strategy design pattern* allows us to use the same methods for both the initial code generation and model synchronization.

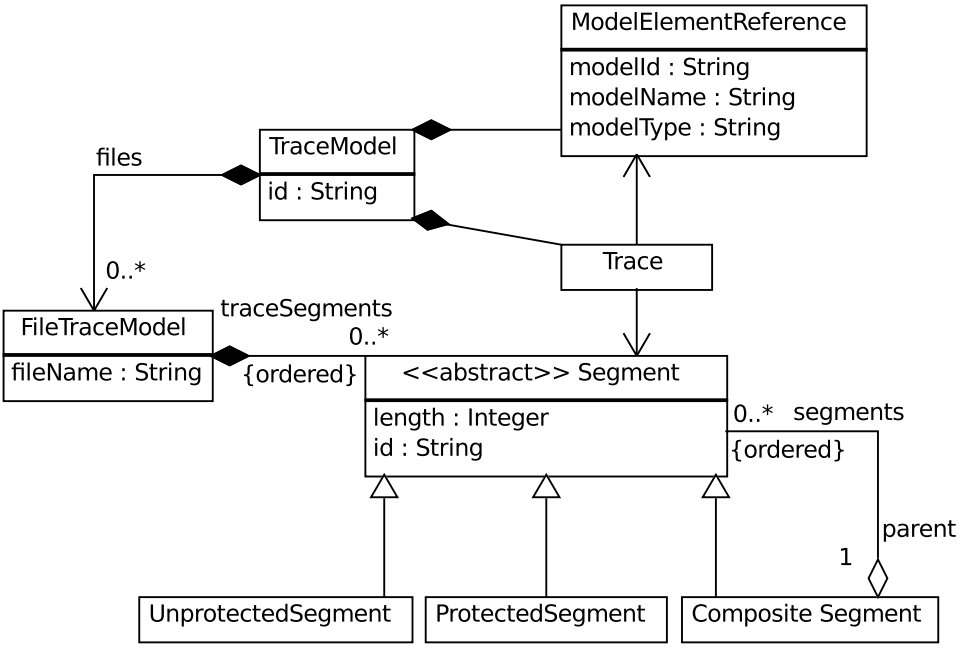

Fig. 3: Trace model

Fig. 3 depicts our trace model, adapted from the metamodel of traces presented in [OO07]. 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 the 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.

Following the strategy design pattern, 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 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. By following the *proxy design pattern*, 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.

To ensure that source code artifacts that directly correspond to a model element are not manually added by users (and thus hold no corresponding modeling element, making the model an inaccurate representation of the source code), we implemented a model violation detection. For each detected violation a feedback note containing the position of the violation, as well as a message describing it, is provided to the user. Model violations are defined in terms of violation rules. A violation rule consists of a model element type, a regular expression that is used to find the violation, a group number that can be used to reference a specific group of the regular expression and a message that describes the violation.

Live Code Editor The live code editor allows multiple users to collaboratively work on the same file at the same time. As it can be seen in Fig. 4, the live code editor widget is divided into three parts. On the left side of the widget, a list of currently active users as well as a file list is displayed. The actual editor is located in the center of the code editor widget. The cursors of remote users are displayed in different colors to all users participating in this live coding phase. For highlighting protected segments in the viewport of the Ace

Fig. 4: Screenshot of the live code editor widget

editor, we use a gray background color. The depicted screenshot shows the development of a frontend component. Here, a tree containing the widget's *HTML elements* is shown on the right side of the editor. The synchronization of the file content among all users, the synchronization between the source code and its traces and the concept of (un)protected segments are integrated into the code editor. While the content of an unprotected segment can be edited, a protected segment is immutable. In order to synchronize the file content among all users, each unprotected segment is individually synchronized on the frontend, using the previously mentioned Yjs library. As protected segments are not editable, they are not synchronized. Because every unprotected segment is synchronized individually, the length of each segment is also synchronized. Thereby, the transformation of source code changes to updates of the trace model defined in Eq. 7 is implicitly performed for all users. Thus, the trace model and source code are implicitly synchronized for all users. The concept of (un)protected segments is implemented by replacing the default command handler of the Ace editor with a *CommandDecorator* component. This decorator handles code changes of the local user and is called before the command is actually executed by the Ace editor and the corresponding Yjs instance of the edited segment is updated. Based on the type of command, it is decided which actual decorator will be executed. We distinguish between navigation, deletion, insertion and other allowed operations. In order to decide to which segment a source code change belongs, a reference to the current active segment is updated in a navigation decorator, every time the local user's text cursor changes. Based on the active segment, it is decided if a source code change is allowed or forbidden, i.e. the operation is performed in an unprotected or protected segment. A deletion operation is performed if the current active segment is not protected and the dimension of the selected text that should be deleted is not out of the bounds of the active segment. Similar checks are performed for insertion operations. The last group of operations consists of commands that do not change the source code and that can be executed without side effects towards the trace model.

On the backend, we extended the CAE's REST API with means for maintaining local Git repositories. When a request for storing a file is received, its content and its file traces are stored and committed to a local repository. Before that, a check is performed to determine

if storing the file is allowed. To prevent conflicts with files that are artifacts of an earlier model synchronization process, the id of the trace model that is updated in each model synchronization process is also included in each file trace model. Thus, each file is assigned to the id of the model synchronization process in which it was created. Even if our frontend is designed to reload files after a model synchronization process, this protection mechanism additionally ensures that the synchronization between files and their models does not accidentally break. As the content and traces of updated files are (first) only stored in local repositories, the GitHub proxy service provides means to push locally committed changes to a remote repository. Possible conflicts with the remote repository are automatically resolved by using the Git *Theirs* merging strategy. This strategy ensures that in case of merging conflicts, changes of the branch that is merged are used for resolving the conflicts.

6 Evaluation

We performed a usability study with student developers to assess how our collaborative MDWE method is received in practice. We carried out eight user evaluation sessions, each consisting of two participants. After receiving a short introduction into the CAE 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. Each evaluation session took about half an hour of development time. At the end of each session, we let the participants fill out a five Likert scale questionnaire containing questions about their Web development experience and gathered their feedback regarding the cyclic development process and the live code editor.

Results and Observations. As expected, the (pre-survey) rating of the familiarity with Web technologies (4.00) and RESTful Web services (4.07) 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). Fig. 5 shows the main results regarding our development paradigm. As it can be observed, 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. The same also holds for the awareness introduced into the live code editor, as the participants could easily see where other developers were working (4.33). They were able to successfully collaborate on a shared part of the application by using the live code editor (4.47). These two rather high ratings show that the developed live code editor fulfills the requirements for live collaborative code editing of model-based applications. Compared to the other ratings, the general idea of a collaborative code editor for development and the need for collaboration during the code refinements phase were rated lower (both 3.47). One explanation for this is that developers are familiar with using version control systems and therefore do not see a high demand for a live collaborative code editor when working together with other developers. Even though the chosen application was, due to the time constraints of a live evaluation setting, quite

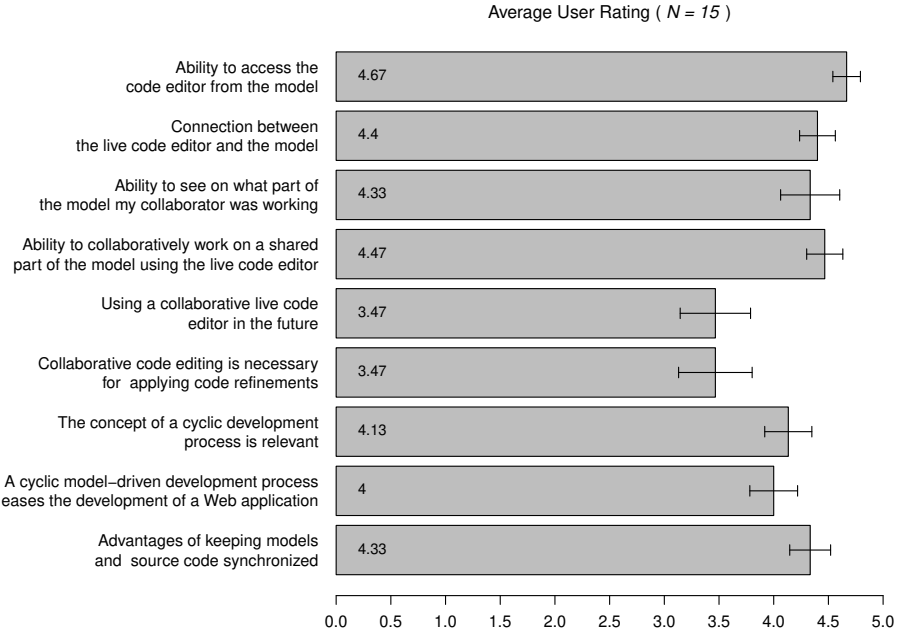

Fig. 5: Results of the user evaluation

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 could identify the advantages of code and model synchronization (4.33). All in all, the perception of participants towards our approach was very positive and we consider it as a successful step towards evaluating our prototype using a more complex application in a real-world development scenario.

In order to evaluate the influence of trace generation on performance and space requirements over time, we measured the code generation time during the complete evaluation and analyzed one frontend component Git repository of the final resulting application. Here, we consider that even though it is isolated, due to our template-based approach, this case gives a good approximation about the behavior of our prototype in various scenarios. Since we used one repository for performing all sessions, we were able to consider data from 490 commits. The exemplary frontend component had a final total size of 634 KB, which contained 9 KB of generated and refined code. The static JavaScript libraries, images and other files added up to 173 KB. The final trace information occupied 42 KB. This leaves 410 KB of Git history, of which the trace history occupied 338 KB. All in all, the trace information, even though it occupies considerably more space than the code output, does not negatively impact the space requirements in a usual Web development setting. Especially, since it only scales with source code changes, which usually don't make up the larger part of an Web

application in comparison to media assets and data. Moreover, the time to store, commit, push and retrieve code with trace information – considering our evaluation scenario data and participant's subjective opinion – does not introduce observable delays in the development process.

7 Conclusions and Future Work

This paper presents a concept for synchronizing models and source code, in the context of an agile MDWE scenario on the Web. A trace model providing linking information between model elements and source code artifacts, as well as a prototype of a live collaborative code editor that supports our concept of traceability and model synchronization have been developed and integrated into an existing MDWE approach. The evaluation of our prototype showed that our method is relevant and a valuable enhancement for introducing agile development practices into MDWE.

As future work, we plan to extend our evaluation on larger scale projects, to gain deeper insights on the effects of using our agile MDWE method. Since the developed prototype is integrated into a larger framework with which we want to integrate complete professional communities into the development process, we want to further investigate the impact which the interplay between live coding, live preview of Web application changes and collaborative modeling has on the communication between end-users and developers.

References

[ALC08] Angyal, László; Lengyel, László; Charaf, Hassan: 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. IEEE Computer Society, pp. 463–472, 2008.

[BK09] Busch, Marianne; Koch, Nora: MagicUWE – A CASE Tool Plugin for Modeling Web Applications. In: Proceedings of the 9th International Conference on Web Engineering. Springer, pp. 505–508, 2009.

[CFB00] Ceri, Stefano; Fraternali, Piero; Bongio, Aldo: Web Modeling Language (WebML): A Modeling Language for Designing Web sites. Computer Networks, 33(1):137–157, 2000.

[CH06] Czarnecki, Krzysztof; Helsen, Simon: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, 45(3):621–645, 2006.

[Go12] Gotel, Orlena; Cleland-Huang, Jane; Hayes, Jane Huffman; Zisman, Andrea; Egyed, Alexander; Grünbacher, Paul; Dekhtyar, Alex; Antoniol, Giuliano; Maletic, Jonathan: The Grand Challenge of Traceability (v1.0). In: Software and Systems Traceability, pp. 343–409. Springer, 2012.

[GW06] Giese, Holger; Wagner, Robert: Incremental Model Synchronization with Triple Graph Grammars. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 543–557, 2006.

[HLR08] Hettel, Thomas; Lawley, Michael; Raymond, Kerry: Model Synchronisation: Definitions for Round-Trip Engineering. In: Proceedings of the First International Conference on Model Transformations. Springer, pp. 31–45, 2008.

[KKK07] Kraus, Andreas; Knapp, Alexander; Koch, Nora: Model-Driven Generation of Web Applications in UWE. In: 3rd International Workshop on Model-Driven Web Engineering. CEUR-WS, 2007.

[La16] de Lange, Peter; Nicolaescu, Petru; Derntl, Michael; Jarke, Matthias; Klamma, Ralf: Community Application Editor: Collaborative Near Real-Time Modeling and Composition of Microservice-based Web Applications. In: Modellierung 2016 Workshopband, pp. 123–127. 2016.

[La17] de Lange, Peter; Nicolaescu, Petru; Klamma, Ralf; Jarke, Matthias: Engineering Web Applications Using Real-Time Collaborative Modeling. In: Proceedings of the 23rd International Conference on Collaboration and Technology (CRIWG 2017). Springer, pp. 213–228, 2017.

[Me02] Mellor, Stephen J.; Scott, Kendall; Uhl, Axel; Weise, Dirk: Model-Driven Architecture. In: Proceedings of the 8th International Conference on Object-Oriented Information Systems. Springer, pp. 290–297, 2002.

[MR03] Martin; Robert Cecil: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, New Jersey, USA, 2003.

[Mv06] Mens, Tom; van Gorp, Pieter: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science, 152:125–142, 2006.

[Ni16] Nicolaescu, Petru; Jahns, Kevin; Derntl, Michael; Klamma, Ralf: Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In: Proceedings of the 19th International Conference on Supporting Group Work. ACM, pp. 39–49, 2016.

[OO07] Olsen, Gøran K.; Oldevik, Jon: Scenarios of Traceability in Model to Text Transformations. In: Proceedings of the Third European Conference on Modelling Foundations and Applications. Springer, pp. 144–156, 2007.

[ORK14] Ogunyomi, Babajide; Rose, Louis M.; Kolovos, Dimitrios S.: On the Use of Signatures for Source Incremental Model-to-text Transformation. In: 17th International Conference on Model Driven Engineering Languages and Systems. Springer International Publishing, pp. 84–98, 2014.

[SR98] Schwabe, Daniel; Rossi, Gustavo: An Object Oriented Approach to Web-based Applications Design. TAPOS, 4(4):207–225, 1998.

[Va14] Vara, Juan Manuel; Bollati, Veronica A.; Jimenez, Alvaro; Marcos, Esperanza: Dealing with Traceability in the MDD of Model Transformations. IEEE Transactions on Software Engineering, 40(6):555–583, 2014.