Providing End-Users with a Visual Editor to Make their Electronic Documents Active

Federico Cabitza, Iade Gesso, Carla Simone
Università degli Studi di Milano-Bicocca
Milan, Italy
{cabitza, gesso, simone}@disco.unimib.it

Abstract—In recent years, visual languages have become increasingly popular in the educational domain. But also researchers involved in other application domains are progressively looking at the potential of visual languages to make difficult tasks, as programming is, easier. In this paper we present the research we have conducted to provide end-users of a document management system with a visual language and the related editor by which to define content-based and context-aware rules. Rules are intended to be the proactive components of electronic documents and the visual rule editor we developed is aimed at empowering end-users and domain experts in making their documents more active with respect to both content, context and the users' interactions.

Index Terms—H.5.2.f Graphical user interfaces, D.2.6.b Graphical environments, D.3.2.t Very high-level languages.

I. INTRODUCTION

Despite the obvious advantages in terms of costs and efficiency that organizations can get from the digitalization of paper-based documents, often this solution does not have the desired effect. As a matter of fact, during the cooperative work that is mediated by paper-based documents, actors naturally develop and share local work practices and conventions that can be disrupted if they are not carefully considered during documents digitalization. If this is the case, the migration from paper-based to digital documents could result in a failure (as illustrated, e.g., in [1]). In the light of this claim during the past years we conducted a series of observational studies and interview sessions with practitioners in some major hospitals of the Northern Italy [2], with the aim to elicit the requirements of increasing autonomy of end-users, we decided to undertake a deep redesign of this component, in order to adopt a purely visual language, and realized a new Mechanism Editor where users can compose rules adopting a building-block metaphor.

To meet these requirements, we conceived the Web Of Active Documents (WOAD) framework [3] and developed ProDoc [4], the first WOAD-compliant prototype; this first application allowed us to perform a preliminary validation of the framework in the health care and in the archaeological domain [5]. WOAD supports users in the autonomous customization of their digital documents in two distinct ways: (i) allowing users to define the structure of their documents using a component based approach implemented in the Template Editor (for more details see [6]); and (ii) allowing users to define, through the Mechanism Editor, a set of proactive rules (called mechanisms) that are executed by an interpreter (namely the Mechanism Interpreter) to convey additional context- and data-related information directly through the digital document itself. This is intended to mimic what happens with textual and graphic annotations that users usually write on paper-based documents to associate a specific situation to specific configuration of signs (or content) that evokes in their mind an appropriate interpretation toward adequate action. The two editors and the interpreter make the WOAD framework belong to the class of environments that support End-User Development [7], [8] explicitly. WOAD proactive rules make the associated documents active [9]. Mechanisms are simple “when-then” constructs: while in the when-part users define the set of conditions to be (asynchronously) checked by the interpreter, in the then-part users specify which kind of additional Awareness-Promoting-Information (API) among the ones identified in [2] has to be proactively provided if those conditions are met, in order to change document affordances to support collaboration awareness and facilitate compliance to coordinative best practices. In its initial definition, WOAD was endowed with a denotational language, called LWOAD [10], allowing users to define mechanisms through a meta-language, then translated in constructs interpreted by the JBoss Drools\(^1\) rule engine [8]. Although we collected some positive evaluations from the practitioners, we realized that LWOAD was still too complex to be fully accepted by users who are not familiar with the constraints that nevertheless are imposed by the underlying formalized language. Then we came to the decision to investigate if a graphical composition of mechanisms could provide a more effective and user friendly interaction; to this aim, initially we realized a wizard-based form by which to create rules following a predefined and step-like process (see [3], [6]). For its closeness to the challenging requirements of increasing autonomy of end-users, we decided to undertake a deep redesign of this component, in order to adopt a purely visual language, and realized a new Mechanism Editor where users can compose rules adopting a building-block metaphor.

Almost all editors that allow the visual definition of rules (e.g., see [11]) are still conceived to be used by ICT professionals and to cope with the requirements of specific applicative domains with little capability to be applied also in others. This is in sharp contrast with our need to make

---

\(^1\)http://www.jboss.org/drools/
end-users autonomous in defining their own rules, and led us to investigate if a more general-purpose visual language can be helpful in meeting this requirement. Visual languages uses range from general purpose software development (e.g., see [12]) to multimedia compositions (e.g., see [13]) and the educational field (e.g., see [14]). With respect to WOAD, none of these visual languages involve the concepts of document and document-related rules, even if some of them adopt a rule-based approach. Moreover, none of these visual languages provide support for collaborative practices, rather they are tools devoted to the development of single-user applications. However, we notice that when simplicity is a key aspect of the editing tool, one of the leading metaphors is that of “building block” (see [15]).

In the following, we will present the work we undertook to implement a visual language (and the related editor) within the WOAD framework, and how it is possible to define some real-life rules, which we identified in the healthcare domain, using the WOAD visual language.

II. THE WOAD MECHANISM EDITOR

In order to have a modular and extensible visual editing solution that is compliant with the WOAD concepts of templates and mechanisms [16] and that could meet the general requirement of empowering end-users and making them autonomous in defining the application logic of their active documents, we chose to customize the solution proposed within the Open-Blocks library [17], an open-source Java library developed at the Massachusetts Institute of Technology to create blocks-based programming user interfaces. This library encompasses a general-purpose visual language in which language constructs (e.g., the different operators) are represented through visual blocks with different shapes. Visual blocks can expose an arbitrary number of strong-typed connectors that can be grouped by two complementary macro-types, i.e., socket and plug, making the composition of complex sequences of blocks more intuitive and less liable to errors.

The WOAD visual language is composed by a small set of simple constructs, which however are sufficient to allow end-users to define the mechanisms they need, even if those mechanisms reach a discrete level of complexity (the complete list of language constructs is reported in Table I). While in the right hand side (RHS) of a mechanism the only allowed constructs are the set actions, through which users specify the APIs they want to convey, the left hand side (LHS) allows users to define conditions using a wider set of language constructs (e.g., comparators, arithmetical operators and aggregators2). The WOAD visual language assumes that all conditions in the mechanism LHS are evaluated as in strictly logical conjunction (i.e., in AND); to define a disjunctive set of conditions (i.e., OR) users have just to define them in terms of distinct mechanisms, containing each one condition to be evaluated.

2Under the concept of aggregator the WOAD visual language includes both a set of SQL-like aggregators (i.e., COUNT and AVG) and the inclusion operator (i.e., IN).

<table>
<thead>
<tr>
<th>Constructs</th>
<th>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>set API b</td>
<td>x</td>
</tr>
<tr>
<td>Aggregation</td>
<td>average, count, in</td>
<td>x</td>
</tr>
<tr>
<td>Arithmetical</td>
<td>+, −, *, /, %</td>
<td>x</td>
</tr>
<tr>
<td>Comparison</td>
<td>=, ≠, &lt;, ≤, c, &gt;, ≥, like d</td>
<td>x</td>
</tr>
<tr>
<td>Logic</td>
<td>not</td>
<td>x</td>
</tr>
<tr>
<td>Rule</td>
<td>mechanism</td>
<td></td>
</tr>
</tbody>
</table>

These columns respectively indicate if a language construct can be used either in the rule left hand side (LHS) or in the rule right hand side (RHS).

API is a placemark for APIs’ names (e.g., set criticality). Only for numbers.

3Only for strings.

In order to build the new Mechanism Editor it has been necessary to map the set of visual language constructs to the OpenBlocks language definition and to integrate the visual editor with the WOAD document templates to automatically populate the blocks palette with the data fields in document templates (which had been previously defined with the Template Editor). Subsequently, the visually defined mechanisms have been made readable by the JBoss Drools engine in the Mechanism Interpreter. To this aim, we defined an intermediate, XML-based language, which is independent of the OpenBlocks serialization format and preserves the mechanism application logic only as it discards all the OpenBlocks topological data. This solution gives us the advantage of having a visual editing solution that is independent of the underlying execution system, and vice versa. In particular, we developed two distinct translators: while the former translates the OpenBlocks language into our intermediate language, the latter translates the intermediate language into the native Drools rule language (DRL) syntax. Moreover, we simplified the OpenBlocks demonstration editor user interface (see Figure 1), in order to have end-users focus on the visual composition of their mechanisms, while minimizing the risk to confuse them with unnecessary graphical elements. For the same reason, when users start the creation of a new mechanism, the WOAD Mechanism Editor automatically places a “mechanism” block in the editing area. Intuitive icons and short pop-up messages have been placed on top of each scrollable panel associated to a tab in the blocks palette in order to provide users with an intuitive description of what the visual blocks that are listed in the panel are about. Finally, in order to help end-users to better contextualize the visual blocks they are using with the document-data those blocks refer to, the interface offers a clickable icon that opens a thumbnail of the related document template in a zoomable preview window.

Leveraging the functionalities provided by the OpenBlocks library, most part of the operations that end-users can perform to build a mechanism are based on drag’n’drop interactions on blocks and on their spatial composition, as if blocks were pieces of a puzzle. Consequently, in the WOAD Mechanism Editor users are called to type something in only to specify...
constant values. Thus, creating a more or less complex mechanism generally requires the user to apply a well defined set of actions. The user selects an element in the blocks palette, opening a panel with a list of blocks, and places the visual block she needs to insert in the mechanism by dragging and dropping it in the edit area. To make a block part of the mechanism, the user will connect it to another block, which is already part of the mechanism, just dragging the block near to the second one. If the two blocks respectively expose a socket and a plug connectors of the same data type, then they will connect to each other. When the user is done with editing, she can either save the mechanism for further modifications, or export it to be executed by the Mechanism Interpreter.

After having developed the new Mechanism Editor, we proceeded with an experimental user study, which is discussed at length in [18] and demonstrated how the editor could be sufficiently simple to use for end-users with no particular programming skills and still be also powerful enough to support the definition of real-life rules.

III. EXAMPLES FROM THE CLINICAL DOMAIN

To better clarify how end-users can use the WOAD visual language to define their own mechanisms, in the following we will present some real-life rules from the healthcare domain, which have been previously reported and discussed in [19].

The first mechanism we designed regards the convention by which doctors are warned whenever they are prescribing a drug that is not present in the list of drugs that has been previously defined for their specific unit (the whole mechanism can be seen in Figure 2). The rule involved two distinct document templates, which are respectively called Drug List Sheet and the Prescription Sheet. To make this rule computational, the end-user (i.e., in this case, either a doctor or a nurse) would perform a simple sequence of operations: picking up the “equals” operator block (i.e., “=” ) in the blocks palette; dragging and dropping the selected block in the editing area; connecting it to the “mechanism” block. Then, the user has to repeat these operations to put the “in” operator in the mechanism; this latter operator checks whether a value is present within a specific set of fields, and the blocks related to the drug name field (in Prescription Sheet) and the set of drug names (in Drug List Sheet). Finally, the user puts the “false” constant value block in connection with the “=” block, in order to complete the condition. In regard to the then-part, users complete the mechanism by adding a specific action block to convey what we called Revision API, i.e., an information by which the system promotes the users’ awareness of the need to revise something: for instance, the Revision API can be defined to draw an icon that represents a yellow exclamation mark near to the field that has been checked, i.e., the drug name field. Similarly, it is possible to define a mechanism by which to check whether a prescribed drug can be dangerous according to some clinical condition, e.g., by checking a list of potentially dangerous drugs so as to convey a Safety API: this action can change the color of the drug name to make clinicians aware of the risk of misdosing that particular drug.

A more complex rule regards the usual check that neonatologists perform immediately after and five minutes after the delivery of a premature newborn to see if her APGAR score is lower than 4, a threshold that is associated with specific procedures; this mechanism involves the document template that is called Newborn Sheet. This mechanism requires to define multiple and complex conditions in the when-part and to specify constant values in these conditions. To this aim, users start defining the first condition by dropping the “greater than” operator (i.e., the “>” block) into the editing area; subsequently, users performed the same operation with the “=” block (i.e., the light orange block in Figure 3), in order to check the result of an arithmetical difference between two values, which they connected with the two blocks that respectively map the current system time and the newborn delivery time. Finally, users completed the condition by connecting a numeric value block (i.e., the dark green block) to the “>” block: at this point, users had just to edit this block to specify its constant value; to this aim they typed this value after enabling the “block editing mode” with a double click on the constant value block. Similarly, users would also define a simpler condition to check if the APGAR data field value is lower than 4. Once users have completed this condition, they drag and connect the condition block under the first one (see Figure 3). Again, users can complete the then-part of the mechanism by associating it to the action block by which the system conveys Criticality API, i.e., an indication that makes aware users of the occurrence of something critical: for instance, the Criticality API can be defined to draw a thick
Fig. 3. The user adds the second condition in the APGAR score example red border around the APGAR field.

IV. Conclusions

The paper illustrated the main tenets of the WOAD framework whose aim is to allow users to keep the coordinative practices and conventions they have defined and shared in their collaboration mediated by documents, once these have been digitized. Specifically, the paper focused on the new Mechanism Editor, i.e., a tool that adopts, extending and simplifying it, the visual language developed within the MIT’s OpenBlocks initiative, in order to allow end-users to define *when-then* constructs that make digital document proactive in evoking in the minds of the cooperating actors good practices and coordinative conventions.

The literature and our empirical work show that cooperative work that is mediated by (digital) documents that are created and progressively tuned by end-users themselves is a field that deserves increasing attention since this is pervasive in multiple domains and suggests natural ways to make the transition from paper-based to digital documents acceptable to the users, given that their situated practices and local, even informal conventions are duly preserved and supported. Our research is continuing on this track towards a general functionality that allows users to collaboratively annotate multimedia documents and supports both the informative and the communicative roles of annotations [20]. In particular, the latter role is crucial in the definition and sharing of conventional interpretations and practices. This annotation functionality will enrich the WOAD framework (and its tools) of end-user oriented capabilities and will be integrated with the creation and management of the computational mechanisms that have been illustrated in this paper.

REFERENCES


